What is C++

Bjarne Stroustrup at Bell Labs developed C++ in late 1980s. It's main purpose is to make writing programs easier for real life problems.

For the last couple of decades C programming language has been widely accepted for all applications. C is powerful structured language. C is reliable, simple and easy to use. C is portable and has ability to extend itself. But it is observed that certain real life problems are difficult to code in C. As the programs grew larger, even the structured approach fails to show the desired result. The programs becomes difficult to maintain and reusability of programs decreases. Bjarne Stroustrup developed C++ based on C. This is the reason C++ is called as incremented version of C or superset of C. Most of the C features are also available in C++. In addition to those, Bjarne stroustrup added features of Object Oriented Programming (OOP). OOP is an approach to program organisation and development. In OOP the emphasis is on data rather than procedure. Programs are divided into objects and it follows bottom up approach in program design.

The approach of OOP is more closer to real life problems. Suppose you want to built a house, or repair your bike, or even to take admission for a course, first you think about the object and its purpose and behaviour. Then you select your tools and procedures. The solution fits the problem.

In real life many times we use the same name but with different reference. As human beings, we understand it. C++ provides constructs so that we can assign same name, to similar actions.

The object - oriented features such as classes, function overloading, operator overloading are added in C++. This makes C++ an Object Oriented Programming Language.

Editors, Compliers. Data bases, Communication Systems, and any real life complex system can be developed in C++. C++ programs can be easily maintained and expanded. A new feature can be easily added in C++ program. In C++. The elements of GUI (Graphical user Interface) such as menus, windows etc can be developed. Now a days the demand for GUI based software is increasing.

C++ is now replacing C as a general purpose language.

Character Set

A character is an alphabet, digit or special symbol. It is used to represent information. Every language allows only certain characters to be used in a program. The collection of all such characters is called character set. Every language has its own character set. When we write a program in a language then we can use only those characters from the character set of the language.

Table 3.1 shows valid alphabet, numbers and special symbols allowed in C++. Observe that the character set of C++ is same as that of C.

Alphabet	A, B, C, D, ------ Z a, b, c, d, ------ z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * ( ) _ – + | \ { } [ ] : ; " < > , . ? /

Table 3.1 : 'C++' character set.

Tokens

The smallest individual units in a program are called tokens. It is a common name given to similar items. C++ has following tokens :

* Keywords
* Identifires
* Constants
* Strings
* Operators

A C++ program consist of these tokens and white spaces. The program is written as per the syntax rules of the language. The syntax rules are grammetical rules for writing programs.

Data types

Data type is the set of quantities which belongs together and are treated similarly. It is a name given to similar quantities. C++ is very rich in its data types. Figure 3.1 shows data types in C++.

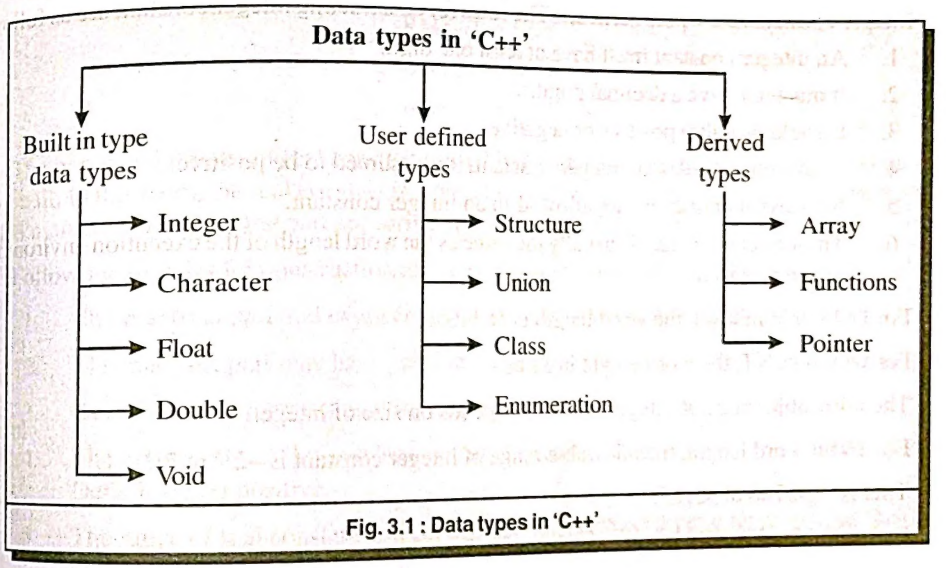


Fig. 3.1 : Data types in 'C++'

Data types in C++ can be classified into three categories.

i) **Built in type**
ii) **User defined type**
iii) **Derived type**

Built in type is also called as Basic or Primary data type. There are five basic data types.

Integer (int), character (char), Float (float), Double (double) and Void (void) are basic data types.

One of the major advantage of C++ is that user defined data types can also be used as if built in data type. Structure (structure), Union (union), Class (class) and Enumeration (enum) are user defined data types.

The user defined data type are defined by user in the programs as per need of the programmer. This feature of adding own data type makes the C++ an extensible language.

The Derived data types are also called as structured data types or secondary data types. They are formed by using basic data types of the language. Array, Function, Pointer are derived data types.

First of all we will discuss built in types of C++. The rest of the data types will be discussed later.

Integer Constant

Lets first understand what we mean by constant and variable in C++. A constant is a quantity that does not change. This quantity can be stored at locations in the memory of the Computer. A variable is a name given to the location in memory where the constant is stored. Naturally the contents of the variable can change.

Integer constants are whole numbers. The rules for constructing integer constant are as follows:

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative.
4. If no sign precedes an integer constant, it is assumed to be positive.
5. No special characters are allowed in an integer constant.
6. The size of an integer is usually the same as the word length of the execution environment of the program.

For DOS or Windows, the word length is 16 bits.

For windows NT, the word length is 32 bits.

The allowable range of integer constant depends on size of integer.

For 16 bit word length, the allowable range of integer constant is -2^{15} to $2^{15} - 1$.

That is, –32768 to 32767.

For example, the valid integer constant are 7

+201

–9322

The invalid integer constants are 7.

+–101

932256

$2

Real Constants

Real constants are often called floating point constants. Real constants are fractional numbers. They could be written in two forms, fractional from and exponential form.

Following are rules for constructing real constants in fractional form :

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No special symbols are allowed within a real constant.

For example, the valid constants are 140.9

–7.2

0.09

The invalid real constants are ,

9322

+–7.2

9A2

The exponential form of real constant is used if the value of the constant is either too large or too small. In this form, the real constant is represented in two parts. The part appearing before 'e' is called mantissa, where as the part appearing after 'e' is called exponent.

Following are rules for construction of real constant in exponential form :

1. The mantissa part and exponent part should be seperated by letter 'e'.
2. The mantissa part may have positive or negative sign.
3. Default sign of mantissa part is positive.
4. The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
5. The range of real constant will depend upon the method used to represent the floating point numbers. The range is quite large. A typical range for real constant is

   – 3.4 e 38 to 3.4 e 38 (i.e. -3.4×10^{38} to 3.4×10^{38}).

For example, the valid real constants are :

4.1 e 32 (i.e. 4.1×10^{32})

5.0 e – 5 (i.e. 5.0×10^{-5})

The invalid real constants are,

4 e 5.1

+– 3 e 2

$ 4 e 2

Character Constant

Any single character from the character set is a character constant. Following are rules for constructing character constant.

i) A character constant is either a single alphabet, a single digit or a special symbol, enclosed within single digit or a single special symbol, enclosed within single inverted commas. Both the commas should point to left.

For example, 'A' is valid character while 'A' is not.

ii) The maximum length of a character constant can be 1 character.

For example, the valid character constants are :

'A'

'I'

'5'

'='

The invalid character constants are :

'A'

'Length'

B

Classes of data

The integer constant occupies 2 bytes (16 bits) in memory for 16 bit machine. Out of 16 bits one bit is used for sign. Hence the range of integers is -2^{15} to $2^{15} - 1$ (i.e. – 32768 to 32767). An integer number having value out of above range can also be used by using class facility of C++ language. In order to provide some control over the range of numbers and the storage space, C++ has three classes of integer storage, namely short integer, integer and long integer. Short and long are also called as **modifires**.

Short integers represents fairly small integer values and requires half the amount of storage than ordinary integers. Long integers requires twice the space in memory than ordinary integers and it represents large set of integer values as shown in table 3.2.

Type	Size (bits)	Range
integer	16	– 32768 to 32767 (i.e. -2^{15} to $2^{15} - 1$)
short integer	8	– 128 to 127 (i.e. -2^7 to $2^7 - 1$)
long integer	32	– 2, 147, 483, 648 to 2, 147, 483, 647 (i.e. -2^{31} to $2^{31} - 1$)

Table 3.2 : Size and Range of integer, short integer and long integer on 16-bit machine.

Integers, signed and unsigned

Some times, we know in advance that the integer constants which we are using will be always positive. In such a case we can use unsigned form of integer constant. Here the range of integer constant gets increased, since we need not to reserve one bit for sign.

The range of unsigned integer is from 0 to 65535 (i.e. from 0 to $2^{16} - 1$).

The default declaration of integer constant will be always a signed number.

Table 3.3 shows range and size of unsigned integers.

Type	Size (bits)	Range
unsigned integer	16	0 to 65535 (i.e. 0 to $2^{16} - 1$)
unsigned short integer	8	0 to 255 (i.e. 0 to $2^8 - 1$)
unsigned long integer	32	0 to 4, 294, 967, 295 (i.e. 0 to $2^{32} - 1$)

Table 3.3 : Size and Range of Unsigned Integers.

Characters, signed and unsigned

Characters are usually stored in one byte (8 bits) of internal storage. Character can have values from – 128 to 127 (i.e. -2^7 to $2^7 - 1$).

In case of unsigned characters this range gets increased from 0 to 255. (i.e. 0 to $2^8 - 1$).

Floats and doubles

Floating point or Real numbers are stored in 4 bytes, (32 bits) of memory. The range of floating numbers is from – 3.4 e 38 to 3.4 e 38.

If this is insufficient then C++ offers a double data type. It occupies 8 bytes (64 bits) in memory and has a range – 1.7 e 308 to 1.7 e 308.

A long double occupies 10 bytes (80 bits) in memory and has a range – 3.4 e 4932 to 1.1 e 4932.

Variables

Variable is a name given to memory location where different constants are stored. These locations can contain integer, real or character constant. The type of the variable depends on the type of the constant it can handle. The rules for constructing variable names of all types are same. These are as given below.

i) The variable name is any combination of alphabets, digits and underscores.

ii) The variable name can not start with digit.

iii) Uppercase and lowercase letters are distinct.

iv) No special symbol other than underscore can be used in variable name.

In C++, there is no limit on the length of the variable name.

For example, the valid variable names are :

i
count
T1
si _ int

The invalid variable names are :

$ 200
L1, L2
1S

Declaration of variables

After designing the suitable variable name, we must declare them to compiler before we use them.

Declaration of variable does the two things.

i) It tells the compiler what the variable name is.

ii) it specifies what type of data the variable will hold.

In C++, a variable can be declared any where in program before its use. It is not necessary that we should declare all the variables at the beginning of program itself. Even we can declare a variable just before its use.

The syntax of declaring the variable is :-

data- type $v_1, v_2, \text{----- } v_n$;

$v_1, v_2, \text{----- } v_n$ are names of variables. They are seperated by commas. The data-type indicates the data type of the value that the variables will hold.

For example,
```
float average ;
int x ;
double ratio ;
```

Here float, int and double are keywords to represent float type, integer type and double data values respectively.

Keywords

Keywords are those words whose meaning has already been explained to the complier. The keywords implements specific feature of the language. The keywords can not be used as variable names. There are 48 keywords for C++ language. They are listed in figure 3.2.

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Fig. 3.2 : C++ keywords.

A detailed discussion of these keywords would be taken up in later chapters whenever their use is relevant.

Here you should note that learning a programming language means understanding the meaning (Semantics) of keywords of language and using them in proper format (Syntax) within the program. When we learns Syntax and Semantics of all keywords of C++ then we will be master of C++.

Hexadecimal and Octal Constants

The number system that we normally use is decimal number system. It has a base of 10. But sometimes it is eaisier to use number system having base of 8 or 16. The number system having base of 8 is octal number system and uses digits 0 to 7. The decimal 8 is 10 in octal. The number system having base of 16 is hexadecimal number system and uses digits 0 to 9, plus the letters A through F. The hexadecimal number 10 is 16 in decimal. The integer constant can be specified in hexadecimal or octal form instead of decimal. The octal number begins with letter o and hexadecimal number begins with letter ox.

For example, ox 8A, is hexadecimal number
o12, is octal number
ox 8 0, is a hexadecimal number.

String Constant

A string is a set of characters enclosed in double quotes.

For example, the valid string constants are :

```
"Hello!"
"This is a string"
```

A string may consist of any character from character set. Observe that A is single character constant and is different from "A", which is a string.

Backslash Character Constants

Backslash character constants are escape sequences. They are not printable characters but they are used in output functions. The list of backslash character constants is given in table 3.4.

For example, '\n' is new line character.

Code	Meaning	Code	Meaning
\b	Backspace	\f	Form feed
\n	New line	\r	Carriage return
\t	Horizontal tab	\"	Double quote
\'	Single quote	\0	Null
\\	Backslash	\v	Vertical tab
\a	Alert	\?	Question mark
\N	Octal constant	\xN	Hexadecimal constant

Table 3.4 : Backslash constants

Operators

An operator is a symbol that tells the computer to perform certain mathematical and logical manipulations. Operators are used in program to manipulate data and variables. C++ has a rich set of operators. They are classified into following categories.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and Decrement operators
6. Conditional operator
7. Bitwise operators
8. Special operators of C++

Arithmetic operators

The operators used to perform arithmetic operations such as addition (+), subtraction (–), multiplication (*) and division (/) are called arithmetic operators. C++ provides all the basic arithmetic operators. They are listed in table 3.5.

Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Table 3.5 : C++ Arithmetic operators

Note that C++ does not have operator for exponentiation. The variables and constants on which operators are operated, are known as operands. The arithmetic operators are unary or binary operators. That is, they require one or two operands.

The arithmetic in C++ can be of three types.

i) Integer Arithmetic
ii) Real Arithmetic
iii) Mixed Mode Arithmetic

Integer Arithmetic

When both the operands are integers, the operation is called integer arithmetic. The integer arithmetic always yields an integer value.

For example,

```
55 – 5   = 50
55 + 5   = 60
14 * 4   = 56
14 / 4   = 3     (Decimal part turncated)
14 % 4   = 2     (Remainder of division)
–14 % 3  = – 2   (Sign of result is
14 % 3   = 2      sign of first operand)
6/7      = 0
```

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in fractional or exponential form.

For example,

```
1.5 – 0.7       = 0.8
1.5 + 0.7       = 2.2
1.5 * 2.0       = 3.0
6.0 / 7.0       = 0.857143
1.0 / 3.0       = 0.333333
– 2.0 / 3.0     = – 0.666667
```

Note that the operator % can not be used with real operands.

Mixed Mode Arithmetic

C++ permits us to mix integer operand with real operand. When one operand is real and other is integer, the expression is called mixed mode arithmetic expression. If either of the operand is of the real type, then only real operation is performed and the result is always a real number.

For example, 15 /10.0= 1.5

Relational operations

In real life, to proceed further we need to make decisions. Decisions play important role in success. To take a decision, we often need to compare between two quantities, and depending on their relation, we can make decision. For example we may need to compare price of two items or age of two persons. Such comparisons can be done with relational operators.

Relational operators are those operators which are used to perform relational operation such as less than, less than or equal to, greater than, greater than or equal to, equal to and not equal to.

Relational operators are listed in table 3.6.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
<=	is greater than or equal to
==	is equal to
!=	is not equal to

Table 3.6 : Relational Operators in C++

The relational operator along with operands is called relational expression. The relational expression always returns a 'true' value or 'false' value. A 'true' value is a non zero value and 'false' value is a 'zero' value.

However in C++ it is possible to define data type bool and the boolean constants true and false.

A simple relational expression has the following form :

```
arithmetic - exp 1   relop   arithmetic - exp 2
```


Where relop is relational operator. The arithmetic expressions may be simple constants, variables or combinations of them. The arithmetic expressions are evaluated first and then the results are compared.

For example, 10 < 20

This relational expression returns a true value.

(x + 5) > 20

This expression returns a true value if (x + 5) is greater than 20 else it returns false value.

Logical Operators

The logical operators are those which are used to perform logical operations such as logical AND, logical OR and logical NOT. They are listed in table 3.7.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	logical NOT

Table 3.7 : Logical Operators

The logical AND and logical OR operators are used when we want to test for more than one condition and make decision.

For example, 10>5 && Y = = 0

An expression of this kind combines two or more relational expressions to form a logical expression. In above logical expression, 10>5 returns a true value, the second expression Y = = 0 returns true if Y is equal to 0 else it returns false. If both the expressions returns true value, only then the logical expression returns true value. If any one of them is false, the expression will return a false value.

The truth table for logical operators is shown in table 3.8.

OP-1	OP-2	OP-1 && OP-2	OP-1 || OP-2	! OP-1
Non zero	Non zero	1	1	0
Non zero	0	0	1	0
0	Non zero	0	1	1
0	0	0	0	1

Table 3.8 : Truth table of logical operators

In logical expression we can combine more than one expression.

For example,

10 > 5 && ! (10 < 9) || 3 < = 4

The result of above expression is true.

Assignment operators

Assignment operators are used to assign the result of an expression or constant to a variable. The assignment operators can be used within any valid expression, the usual assignment operator is '='. This operator has the form :

Variable-Name = constant or expression;

The arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language.

For example,

```
x = a + b ;
z = 0 ;
final_value = 100 ;
```

During the assignment operation the value of the expression on right hand side is computed and is assigned to the variable on left hand side.

Type conversions in Assignments

When variables of one type are mixed with variable of another type, a type conversion will occur. In assignment statement, the value of the right side of the assignment is converted to the type of the left side. This may involve turncation when real value is converted into integer.

For example,

```
int x;
char ch ;
float f ;
ch = x ;
x = f ;
f = ch ;
f = x ;
```

In ch = x, the high-order 8 bits of integer variable x are lopped off, leaving ch with the lower 8 bits.

If x is between 255 and 0 then ch and x will have identical values. Otherwise, the value of ch would reflect only the low-order 8 bits of x.

In x = f, x will receive the integer part of f. In f = ch, f will convert 8 bit integer value stored in ch to the same value in the floating-point format. In f = x, f will convert integer value of x to the same value in the floating-point format.

Initialization of variables

It is possible to assign a value to the variable at the time the variable is declared. This has the following form :

Data-type variable-name = constants ;

The process of assigning initial values to variables is called initialization.

For example,

```
char response = 'Y' ;
float ratio = 0.5 ;
```

C++ permits us initialization of more than one variable, in one statement using **multiple assignment** operators.

for example,

```
p = q = s = 0 ;
x = y = z = 1.5 ;
```

In professional programs, variables are frequently assigned common values using this method.

Short hand assignment operators

In addition to the usual assignment operator '=', C++ has short hand assignment operators of the form :

Variable OP = expression ;

Where OP is C++ binary arithmetic operator. The operator OP= is known as short hand assignment operator.

For example,	`x += y + 1 ;`	is equivalent to x = x + (y + 1)
	`a += 1 ;`	is equivalent to a = a + 1
	`a -= 1 ;`	is equivalent to a = a – 1
	`a % = b ;`	is equivalent to a = a % b
	`a *= b ;`	is equivalent to a = a * b
	`a /= b ;`	is equivalent to a = a / b

Shorthand notations are widely used in professionally written C++ programs.

Increment and Decrement operators

C++ has two unary operators called Increment and Decrement operators. These are very useful operators used for adding one and subtracting one from operand. The increment operator is '++' which adds one to the operand.

For example, ++ m or m ++

is same as m = m + 1

The decrement operator is '– –' which subtracts one from the operand.

For example, – – m or m – –

is equivalent to m = m – 1.

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand.

For example, m = m + 1 can be written as

++ m or m ++

However, there is difference between prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression.

If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it.

For example,
```
x = 10;
y = ++x ;
```
Sets y to 11. However if you write the code as
```
x = 10;
y = x++ ;
```
y is set to 10. Either way x is set to 11; the difference is in when it happens.

It is always better to use increment and decrement operator whenever we can. C++ generates efficient object code for increment and decrement operators.

Bitwise operators

One of the C++'s powerful features is the set of bit manipulation operators. These permits the programmer to access and to manipulate individual bits within piece of data. Bitwise operators can be used with char and int data types. They can not be used with other data types such as float, double etc. The bitwise operators are listed in table 3.9.

Operator	Meaning
~	one's complement
>>	right shift
<<	left shift
&	bitwise AND
|	bitwise OR
^	bitwise XOR (exclusive OR)

Table 3.9 : Bitwise operators.

One's complement operator

One's complement of a number means replacing all 1's by 0's and all 0's by 1's.

For example, 1's complement of 1010 is 0101. The one's complement operator ~ obtains one's complement of operand.

For example,
```
int i, j ;
j = 3 ;
i = ~ j ;
```
The value of j is 3. In binary form it is 0000 0000 0000 0011.

The value of i will be 1111 1111 1111 1100.

This operator is used to encrypt the file.

Right Shift operator

The right shift operator operates on a single variable. This operator is represented by >> and it shifts each bit in the operand to the right. The number of places the bits are shifted depends on the number following the operand.

For example, x >> 3 would shift all the bits in x three places to the right.

If x contains 11 010111

then x >> 1 would give 0 1101011

and x >> 2 would give 00 11 0101.

When bits are shifted to right, blanks are created on left. They are always filled with zeros.

Observe that shifting operand one bit to right is same as dividing by 2.

For example, 64 >> 1 gives 32

27 >> 1 gives 13

Bit shift operations can be very useful when you are decoding input from an external device, like D/A converter and reading status information.

Left shift operator

This operator is represented by << and it shifts each bit in operand to left and 0 is added to right of the number.

For example, if x = 3 ; i.e. 0000 0011

then x << 1 would give 0000 00110

Observe that shifting operand one bit to left is same as multiplying by 2.

Bitwise AND operator

This operator is represented by & and it operates on two operands. Both the operands must be of same type i.e. either char or int. The & operator operates on a pair of bits to yield a resultant bit.

The truth table is given in table 3.10.

First bit	Second bit	First bit & second bit
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.10 : Truth table of bitwise AND

Bitwise AND operator is used to clear a bit. That is, any bit that is 0 in either operand causes the corresponding bit in the outcome to be set to 0.

for example, (11 00 0001) & (0 111 1111) = 0100 0001

Often parity is indicated by eighth bit, which is set to 0 by AND ing it with a byte that has bit 1 through 7 set to 1 and bit 8 set to 0, as shown above.

Bitwise OR operator

This operator is represented by | and it performs as per following truth table 3.11. It also requires two operands

First bit	Second bit	First bit | second bit
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.11 : Truth table of bitwise OR

The bitwise OR works to reverse to that of bitwise AND. It can be used to set bit to 1.

Any bit that is set to 1 in either operands causes the corrosponding bit in the outcome to be set to 1.

For example, (1000 0000) | (0000 0011) = (1000 0011)

Bitwise XOR operator

The XOR operator is represented by ^ and is also called exclusive OR operator. It also requires two operands. When both the bits are different, it return 1. When both the bits are same, it returns 0. The truth table of XOR operation is shown in table 3.12.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.12 : Truth table of XOR operator

For example, (1111 0000) ^ (0000 0000) = (1111 0000).

Bitwise operations most often find application in device drivers such as modem programs, disk file routines and printers routines. Bitwise operations can be used to mask off certain bits, such as parity. The parity bit is often used to confirm that the rest of the bits in the byte are unchanged.

Also observe that relational and logical operators always produce a result that is either true or false, whereas bitwise operations may produce any arbitrary value in accordance with the operation.

Conditional expression

C++ contains a very powerful and convenient operator that can be used to form conditional expression.

This operator called ternary operator (?:) has the form :

```
exp 1 ? exp 2 : exp 3 ;
```


Where exp 1, exp 2 and exp 3 are expressions. The above expression says "if exp 1 is true (i.e. if its value is non zero), then the value returned will be exp 2 otherwise the value returned will be exp 3".

For example, `y = x > 9 ? 100 : 200 ;`

y is the assigned the value 100 if x is greater than 9, otherwise y is assigned 200.

The above expression is equivalent to

```
if  (x > 9)
then  y = 100 ;
else  y = 200 ;
```

The limitation of the conditional operator is that after ? or after the : only one C++ statement can occur.

The operators which we have discussed so far, are also available in 'C'. But C++ consists of certain additional operators which are not available in C. These operators are discussed below.

<< Insertion operator

<< is insertion or put to operator. It inserts (or sends) the contents of the variable on its right to the object on its left. This operator is used along with cout stream.

For example, `cout << string ;`

This statement will display the contents of string. Here cout is predefined object that represents standard output stream in C++.

>> Extraction operator

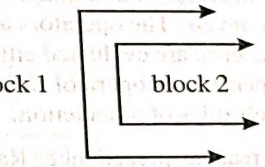
>> is extraction operator or get from operator. It extracts (or takes) the value from the standard input device (keyboard) and assign it to the variable on its right. This operator is used with c in object.

For example, `cin >> number1 ;`

This statement causes the programmer to wait for the user to type in a number. The number keyed in is placed in the variable number1. The cin is predefined object in C++ that corrosponds to standard input stream. Here, it represents keyboard.

:: Scope resolution operator

A C++ program may contain a block within a block .



In C++, if a variable is declared in an inner block (i.e. block - 2) then it hides declaration of same variable is an outer block (i.e. block - 1). Therefore each declaration will cause refer to a different data object.

:: scope resolution operator can be used to uncover the hidden variable. It has the form :

```
:: variable-name
```

The scope resolution operator allows access to a global version of variable.

For example, :: count means global version of count and not the local variable count.

The scope resolution operator is also used in a class member function definition.

New and delete (Memory Management Operators)

C++ has two unary operators new and delete.

The new operator performs the task of allocating memory dynamically at run time.

The delete operator is used to free memory.

Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are ends andl setw.

The endl operator causes a line feed to be inserted.

For example.

```
cout << "m =" << m << endl ;
cout << "n =" << n << endl ;
```

This would cause two lines output.

m = 971

n = 12

The setw operator specifies field width.

For example,

```
cout << setw (5) << sum << endl ;
```

Here the field width is 5 for printing the value of the variable sum.

Precedence and order evaluation

Each operator in C++ has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and operator may belong to one the levels. The operators at higher level of precedence are evaluated first. The operators of same precedence are evaluated either from left to right or from right to left. This is known as associativity property of an operator. Table 3.13 provides a complete list of operators, their precedence levels and their rules of association.

The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and Rank 16 the lowest. Observe that some of the operators in the list are not discussed so far. They will be discussed in later chapters.

Operator	Description	Associativity	Rank
::	Scope resolution operator	Left to Right	1
( )	Function call		
[ ]	Array element Reference	Left to Right	2
.	structure or class member Reference		
->	Pointer to structure or class member reference		
!	Logical NOT	Right to Left	3
+	unary plus		
-	unary minus		
++	Increment		
--	Decrement		
~	One's complement		
*	Pointer Reference		
&	Address		
Sizeof	Size of object		
(type)	Type conversion		
new	New operator		
delete	Delete operator		
*	Multiplication	Left to Right	4
/	Division		
%	Modulo division		
+	Addition	Left to Right	5
-	Subtraction		
<<	Left shift	Left to Right	6
>>	Right shift		
<	Less than	Left to Right	7
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to Right	8
!=	Inequality		
&	Bitwise AND	Left to Right	9
^	Bitwise XOR	Left to Right	10

Operator	Description	Associativity	Rank
|	Bitwise OR	Left to Right	11
&&	Logical AND	Left to Right	12
||	Logical OR	Left to Right	13
? :	Conditional statement	Left to Right	14
= += -= *= /= %= >>= <<= &= ~= |=	Assignment operators	Right to Left	15
,	Comma operator	Left to Right	16

Table 3.13 : Operator precedence and associativity

Lets understand how arithmetic expressions are evaluated in C++.

For example, consider the expression

x = 30/2 + 5 * 4 + 22 – 10

This expression involves +, –, *, / operators. As per precedence rule * and / have higher precedence over + and –. The step wise evaluation is shown below :

x = 30/2 + 5 * 4 + 22 – 10
= 15 + 5 * 4 + 22 – 10
= 15 + 20 + 22 – 10
= 35 + 22 – 10
= 57 – 10
= 47.

Consider another relational expression

x = 10 + 15 && y < 10

As per precedence rule, + operator has higher precedence over relational and logical operators. Suppose x = 20 and y = 5. The step wise evaluation of expression is shown :

x = 10 + 15 && y < 10
x = 25 && y < 10
x = 25 && TRUE
FALSE && TRUE
FALSE

Observe that the final result of expression is FALSE. i.e. the expression will return a zero value.

Type conversions in Expressions

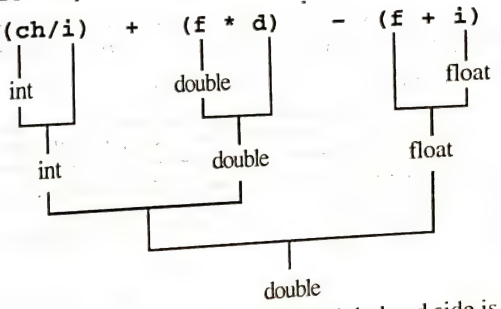
C++ permits us mixing of constants and variables of different types in an expression. When constants and variables of different types are mixed is an expression, then they are all converted to the same type. The compiler converts all the operands up to the type of the largest operand. This is known as type promotion. The type conversion takes place as per following algorithm :

1. First, all char and short int values are automatically elevated to int. This process is called integral promotion.
2. If an operand is long double
   then second operand is converted to long double
   else if an operand is double
   then the second is converted to double
   else if an operand is a float
   then the second is converted to float
   else if an operand is an unsigned long
   then the second is converted to unsigned long
   else if an operand is long
   then the second is converted to long
   else if an operand is unsigned int
   then the second is converted to unsigned int
3. There is one special case. If one operand is long and the other is unsigned int, and if the value of unsigned int can not be converted to long, then both the operands are converted to unsigned long.

Once these conversion rules have been applied, each pair of operands is of same type and the result of operation is the same as the type of both operands

For example, consider

```
char ch ;
int i ;
float f ;
double d ;
x = (ch/i) + (f * d) - (f + i)
```



The type conversion takes place as shown above. The final value on right hand side is double.

Type casts

You can force an expression to be of specific type by using cast. The general form is,

(type) expression

Where type is a valid data type.

For example, (float) x/7

This ensures that type of expression x/7 is float.

Standard I/O streams in C++

To perform input/output operations i.e. to read data from standard input device like keyboard and to write data to standard output device like screen, we need to include iostream.h file in C++ program. The iostream.h is the standard header file input and output stream which contains a set of small and specific general purpose functions for handling input and output data. The I/O stream is a sequence of characters which are to be read from keyboard or to be displayed on screen. The standard input and output operations in C++ are normally performed by using I/O stream such as cin for input and cout for output.

cin

The cin is used to read a number, a character or a string of characters from a standard input device, normally the keyboard. The extraction operator (>>) is used along with cin.

```
cin >> variable 1 >> variable 2 >> ----- variable n ;
```

For example,

```
cin >> a >> b ;
```

cout

The cout is used to display an object onto the standard device, normally the video screen. The insertion operator (<<) is used along with cout.

The general form of cout is.

cout << variable 1 << variable 2 << ----- variable n ;

For example,

```
cout << x << y ;
```

On screen of x and y are displayed. Observe that insertion operator << will not add any spaces between any two data items.

for x = 123 and y = –45, the statement

cout << "x=" << x << '\t' << "y=" << y ;

causes the output : `x = 123 y = - 45`

Observe that message in double quotes in printed as it is and '\t' introduces certain amount of space between two values.

Preprocessor directive

C++ provides a # define directive to define symbolic names. The # define is a directive to the C++ preprocessor. The # define directive causes a symbolic name to become defined as a macro. A macro is associated with a body. The general form of simple macro definition is

 # define macro_name body of macro

For example : #define PI 3.14159265

Here we are assigning a value of Π to symbolic name PI. Observe that there is no semi colon at the end of # define directive. The body of macro can be constants or expression. Previously defined macros can also be used in the definition of a macro. A macro name is any valid variable name. By convention, macro names are written in upper case letters. A # define directive can appear anywher in the program. However, all # define directives are usually collected together at the begining of the program.

C++ preprocessor conceptually processes the source text of a C++ program before the compiler parses the source program. The preprocessor replaces every occurance of a macro name in the program text with a copy of the body of the macro.

Some other examples of simple macros are :

```
# define STEP_SIZE    10
# define EOF          -1
# define MAXINT       214783647
```

Structure of C++ Program

A typical C++ program contains six sections as shown in Figure 3.3.

Include files
Class declaration
Class functions definitions
Non member function prototypes
Main function program
Non member function definitions

Fig. 3.3 Structure of C++ program

These sections may be placed in a seperate code files and then complied independently or jointly.

The header files with .h extensions are included in program at the beginning. The class which are used in program are declared and defined before the main function program. It is also possible to declare all class declarations in a header file and definitions of member functions into another file. Finally the main program which uses the class includes the previous two files.

The non member functions prototypes are declared before main ( ) function and they are defined after the main ( ) function.

A Sample C++ Program

Let us consider the following sample program.

```
//Sample program
# include < iostream.h >
void main ( )
{
  cout << "C++ is better than C" ;
}
```

The above program displays the message "C++ is better than C" on screen.

The first line of program is a comment line. A double slash // is used for single line comment. For block comment we can use / * ----- */. Comments are not processed by compiler.

A second line is # include statement. The header file iostream.h is included in program. This file contains declaration for cout and operator <<. The third line is main ( ) function. The body of main functions starts with opening brace { and gets closed with closing brace }. Within body, each statement is seperated by semicolon. In C++ every function should return a value. If it is not returning any value, then it is declared as void.

We can type in a program by using any standard editor. The program can be compiled and executed by standard C++ compiler such as Turbo C++ compiler.

Simple Programs in C++

Program 3.1

Write a program in C++ to read three integer numbers and compute and display their average.

Program

```
// 3.1 Average of three numbers

#include<iostream.h>

void main()

{
 int m1,m2,m3;
 float avg;

 cout<<"enter 3 numbers";
 cin>>m1>>m2>>m3;
 avg=(m1+m2+m3)/3;

 cout<<"Average="<<avg;
}
```


Output

```
Enter 3 numbers 23 56 67
Average = 48
```

Program 3.2

Write a program in C++ which describes scope resolution operator.

Program

```
// 3.2   :: scope resolution operator

#include<iostream.h>

int a=10;

void main()

{
 int a=15;
 cout<<"\n local a="<<a<<"global a="<<::a;

 ::a=20;
 cout<<"\n local a="<<a<<"global a="<<::a;

}
```

Output

local a = 15 global a = 10
local a = 15 global a = 20

Program 3.3

Write a program in C++ to calculate simple interest by formula.

$Si = \frac{pnr}{100}$, Given p = 1000, n = 3, r = 8.5.

Program

```
/* 3.3  Calculation of simple interest */

#include  <iostream.h>

void main()

{
   int p,n;
   float r,si;
```

```
    p=1000;
    n=3;
    r=8.5;

    si=p*n*r/100;

    cout<<si;

}
```

Output 255

Problem 3.4

Write a program in C++ to find area and perimeter of a rectangle by using formula.

area = pq.

and perimeter = 2 (p + q)

Where p = 4 and q = 6

Program

```
/* 3.4  Area and Perimeter of rectangle */

#include <iostream.h>

void main()

{
  int p,q,area,perimeter;

 p=4;
 q=6;

 area=p*q;
 perimeter=2*(p+q);

cout<<"\n area = "<<area;
cout<<"\n perimeter = "<<perimeter;

}
```

Output area = 24

perimeter = 20

Problem 3.5

Write a program in C++ convert a given temperature in celsius to Fahrenheit by using formula.

$F = 1.8 \text{ celsius} + 32$

Read input celsius through keyboard.

Program

```
/* 3.5 Celsius to Fahrenheit conversion */

#include <iostream.h>

void main()

 {
   float f,celsius;

   cin>>celsius;
   cout<<"\n celsius = "<<celsius;

   f=1.8*celsius+32.0;

   cout<<"fahrenheit = "<<f;

 }
```

Output 56

celsius = 56

Fahrenheit = 132.800003

Problem 3.6

A five digit integer is given. Write a program in C++ to find sum of individual digits. If given number is 16785 then required sum is

1 + 6 + 7 + 8 + 5 = 27

Program

```
/* 3.6 Sum of digits   */

#include <iostream.h>

void main()

{
  int d1,d2,d3,d4,d5,sum,number,n;

  cin>>number;
  cout<<"\n number = "<<number;
```

```
    n=number;
    d1=n%10;
    n=n/10;
    d2=n%10;
    n=n/10;
    d3=n%10;
    n=n/10;
    d4=n%10;
    n=n/10;
    d5=n;
    sum=d1+d2+d3+d4+d5;

    cout<<"\n sum = "<<sum;

  }
```

Output 16785
number = 16785
sum = 27

Problem 3.7

Ramesh's basic salary is input through the keyboard. His dearness allowance is 40% of basic salary and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.

Program

```
// 3.7   Gross salary computation

#include <iostream.h>

void main()

{
   int basic;
   float da,hra,gross;

   cin>>basic;

   da=0.4*basic;
   hra=0.2*basic;
   gross=basic+da+hra;

   cout<<"\n da = "<<da<<" hra ="<<hra<<" gross
   ="<<gross;

}
```


Output 6000
da = 2400
hra = 1200
gross = 9600

Problem 3.8

A projectile fired at an angle θ with an initial velocity v travels a distance d given by

$$d = \frac{v^2}{g} \sin 2\theta$$

Where g is accceleration constant of 9.8 m/sec^2. It stays in motion for a time t given by

$$t = \frac{2v}{g} \sin \theta$$

and attains a maximum height h given by

$$h = \frac{v^2}{g} \sin \theta$$

Write a in C++ to compute d, t and h.

Program

```
// 3.8 Computation of d,t, and h

#include<iostream.h>
#include<math.h>

#define G                              9.8
#define PI                       3.14159265
#define RADIANS_PER_DEGREE   (PI/180)

void main()

{

  float velocity,theta;

  cin>>velocity>>theta;

  /* convert degree into radians */
  theta *= RADIANS_PER_DEGREE;

  cout<<"\n distance="
      <<velocity*velocity*sin(2*theta)/G;
  cout<<"\n time= "<<2*velocity*sin(theta)/G;
  cout<<"\n height= "<<velocity*velocity*sin(theta)/G;

}
```

Control structures

A statement is a part of the program that is executed. A statement specifies certain action. C++ program is a set of statements. These statements are normally executed one after the other. This is known as sequential execution.

But in practise we find sequential execution is always not suitable. There are situations where we need to change our actions as per changing circumstances. Some times we may need to repeat the actions. This involves a kind of decision making. Writing a book is a painful job. Every time I think I may not write further. But looking at its creativity, I go for it.

While coding for real life problems, we may need to take decision and change our action. For this C++ provides two control structures.

1. Selection structure (Branching)
2. Loop structure (Iteration or Repitition)

The selection structure is also called as branching, or conditional structure. Here we test for a condition and based on result of condition we execute certain group of statements. A group of statements is also called as block statement. Sometimes without checking for any condition we may jump to another statement. C++ provides following statements to achieve this.

1. The If statement
2. The If - Else statement
3. The Else - if statement
4. Switch statement
5. Goto statement

Loop structure is also called as Iteration or repitition. In loop structure, certain group of statements is executed repeatedly by forming a loop. The loop is executed for definite number of times.

C++ provides following statements for formation of loop.

1. While statement
2. Do - while statement
3. For statement

The Break, continue, goto and return statements are used to come out of the loop.

Now we will discuss the conditional and loop control statements of C++ along with their Syntax and Semantics.

Simple If Statement

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is a two way decision statement and is used in conjunction with an expression. The general form of if statement is shown .

```
if (test expression)
{
   statement - block ;
}
statement - x ;
```


Here if is a keyword. The test-expression is a expression. This expression always returns a true value (i.e. non zero value) or a false value (i.e. a zero value). The statement - block may be a single statement or a group of statements where each statement end with semicolon. The statement - x is the next statement appearing after statement - block.

In if statement, first of all test expression is evaluated. If it is true then statement - block is executed, otherwise the statement block is skipped and the execution will jump to statement - x. Observe that when condition or test - expression is true, both the statement - block and statement - x are executed. But if condition is false, only statement - x is executed. The flow chart of simple if statement is shown in figure 3.4.

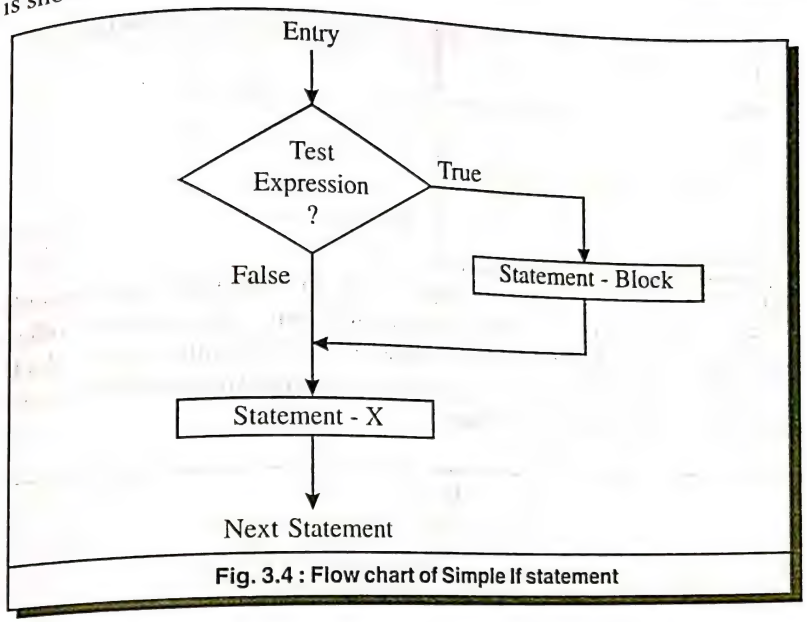


Fig. 3.4 : Flow chart of Simple If statement

Let us consider some programming situations where you need to use if statement.

1. Suppose we are writing program to implement discount policy. If quantity is more than 1000 then discount is given in rate. For this we write if statement as :

```
if (quantity > 1000)
{
   rate  = rate - discount ;
}
   cout << rate ;
```

2. If a student is belonging to sports category, then an addition of bonus-marks is made to his marks. For this we write if statement as :

```
if (category = = sports)
{
   marks = marks + bonus_marks ;
}
   cout << marks ;
```

The if - else statement

The if - else statement is an extension of simple if statement. In simple if, if the condition is false then we go to next statement. Many times we need to execute certain group of statements if the condition is false. In such a case we need to use if - else rather than simple if.

The general form of if - else statement is shown.

```
if (test expression)
{
    True - block statements;
}
else
{
    False - block statements ;
}
statement - x ;
```

Here if and else are keywords. The test - expression is an expression. It returns either a true or false value. If test expression is true, then true-block statements are executed ; otherwise false-block statements are executed. In either case, either true - block statements or false - block statements are executed, not both. In both the cases the control is subsequently transferred to statement - x. The flow - chart for if - else is shown in figure 3.5.

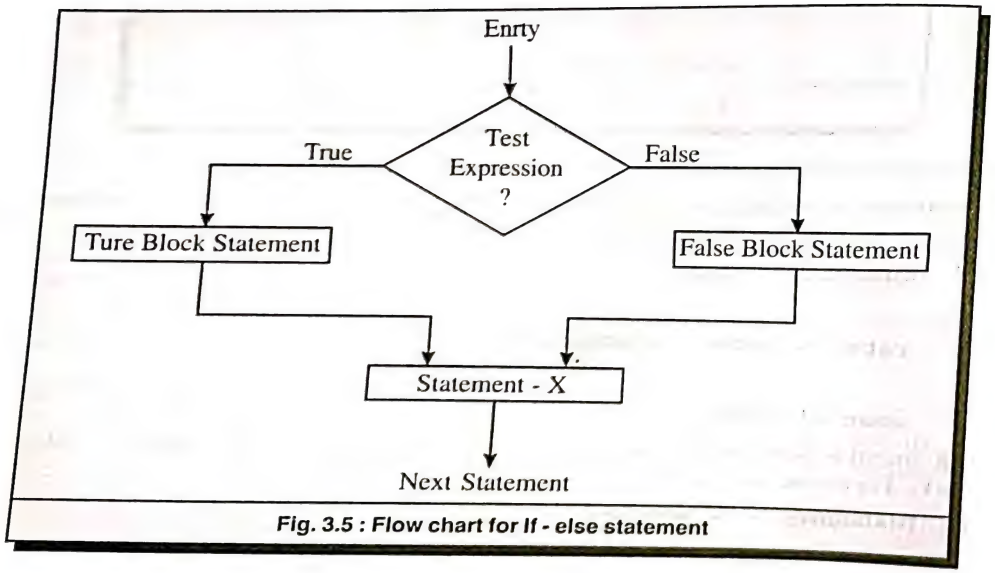


Fig. 3.5 : Flow chart for If - else statement

*Now let us consider some programming situations where you need to make use of **if - else**.*

1. *Suppose we are writing program to find largest of two numbers. First, we compare between first two, then depending on result we decide which is larger. The **if - else statement** for this situation is shown below.*

```
if(a > b)
  {
       // True - block statements
       cout << "Largest=" << a ;
  }
  else
  {
       // false - block statements
       cout << "Largest="      << b ;
  }
```

2. *Suppose we are counting for males and females. We check for code, then depending on result we increment the counter for male or female. This is shown by following **if - else statement**.*

```
if(code = = 1)
  {    // True - block statements
       males = males + 1 ;
  }
  else
  {    // false - block statements
       females = females + 1 ;
  }
```

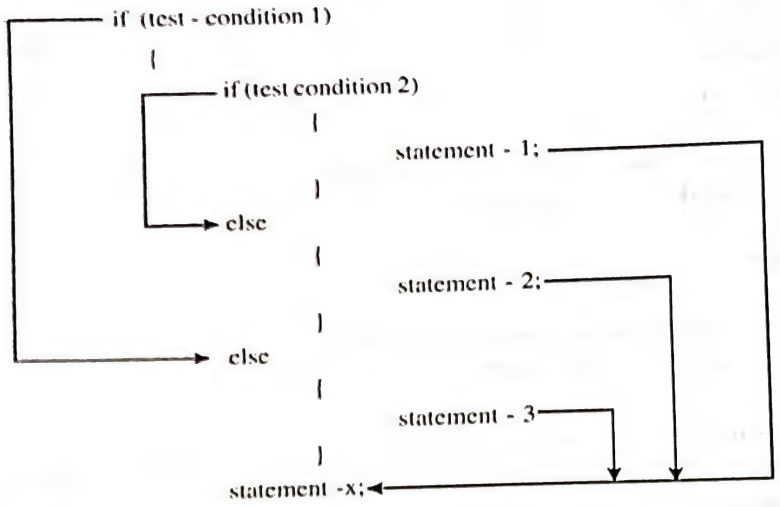
Note that if there is only one statement to be executed after test condition, then we may not use opening brace ({) and closing brace (}).

```
if   (test expression)
     statement A ;
else
     statement B ;
```

In C ++ the test expression must produce either an integer, character, pointer, floating - type or bool type result.

Nested – ifs

In practice, many times we need to make a series of decisions. In such a case we may have to use more than one if - else statement in the nested form. A nested if is an if that is the target of another if or else. In nested if, an else always refers to the nearest if statement that is within the same block.



Here first of all condition - 1 is evaluated. If the condition - 1 is false, statement - 3 will be executed; otherwise condition - 2 is tested. If condition - 2 is true, statement - 1 is executed otherwise statement - 2 is executed and control is transferred to statement - x. The flow chart is shown in figure 3.6.

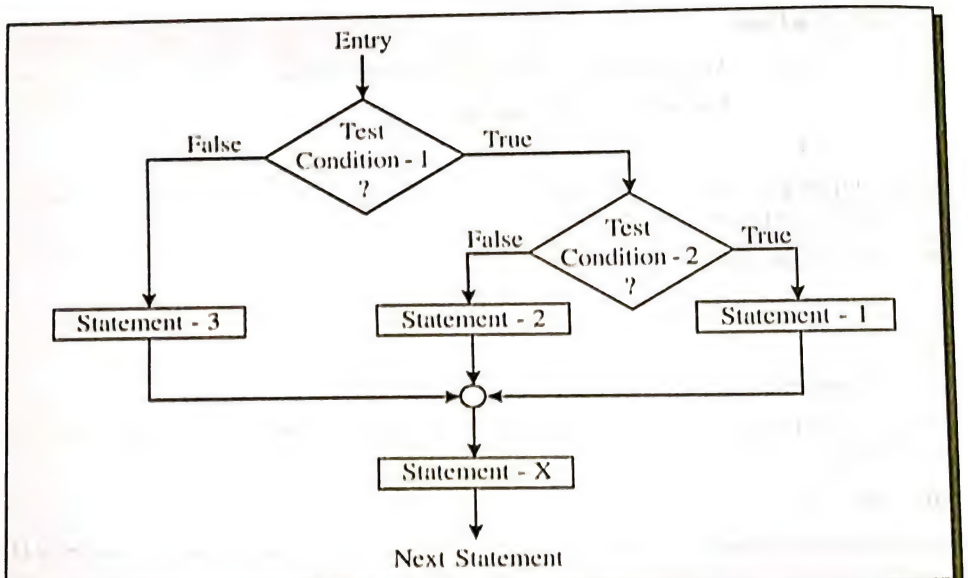


Fig. 3.6 : Flow chart for Nested - If statement

When nesting care should be taken to match every if with an else statement. An else is linked to the closest non-terminated if. That is else is associated with inner if. In C++, there is no limit on nesting of if-else.

Now let us consider some programming situations where you need to make use of nested - ifs

1. Suppose a bank has policy of giving additional bonus of 5% for female candidates having balance more than 10000, otherwise a bonus of 2% will be given. This policy can be implemented by nested its as follows.

```
If (sex = = 'f' )
   if (balance > 10000)
     bonus = 0.05 * balance ;
   else
     bonus = 0.02 * balance ;
```

Observe that here there are two if's and one else. The question is with which if the else is associated with. This else is called as dangling else. This else is associated with second if.

But if we write

```
If (sex = = 'f' )
 {
   if (balance > 10000)
     bonus = 0.05 * balance ;
 }
 else
     bonus = 0.02 * balance ;
```

The policy gets changed. Here the bonus for female candidates having balance less than or equal to 10,000 is not calculated.

2. Consider another situation when we are finding largest among the three numbers. We first compares between the first two, depending on result we compare first and third or second and third, to determine largest. This can be implemented as shown below.

```
if  (a > b)
 {
    if (a > c)
      cout << "Largest=" << a ;
    else
      cout << "Largest=" << c ;
 }
 else
```

```
{
   if (b > c)
     cout << "Largest=" << b ;
   else
     cout << "Largest="  << c ;
}
```

The else - if ladder

When there are multipath decision involved, we can use if's, another way. Here we have a chain of ifs and each else is associated with if statement. This programming construct is called else - if ladder. Because of its appearance, sometimes it is also called as if - else - if staircase. The general form of else - if ladder is

```
if (condition - 1)
    statement - 1 ;
else if (condition - 2)
    statement - 2 ;
else if (condition - 3)
    statement - 3 ;
    ----------
    ----------
else if (condition - n)
    statement - n ;
else default statement ;
statement - x ;
```

The conditions are evaluated from top, downwards. As soon as a true condition is found, the statement associated with it is executed and control is transferred to statement - x, skipping the rest of the ladder. When all n - conditions becomes false, then the final else containing the default statement is executed. The flow chart for else - if ladder is shown in figure 3.7.

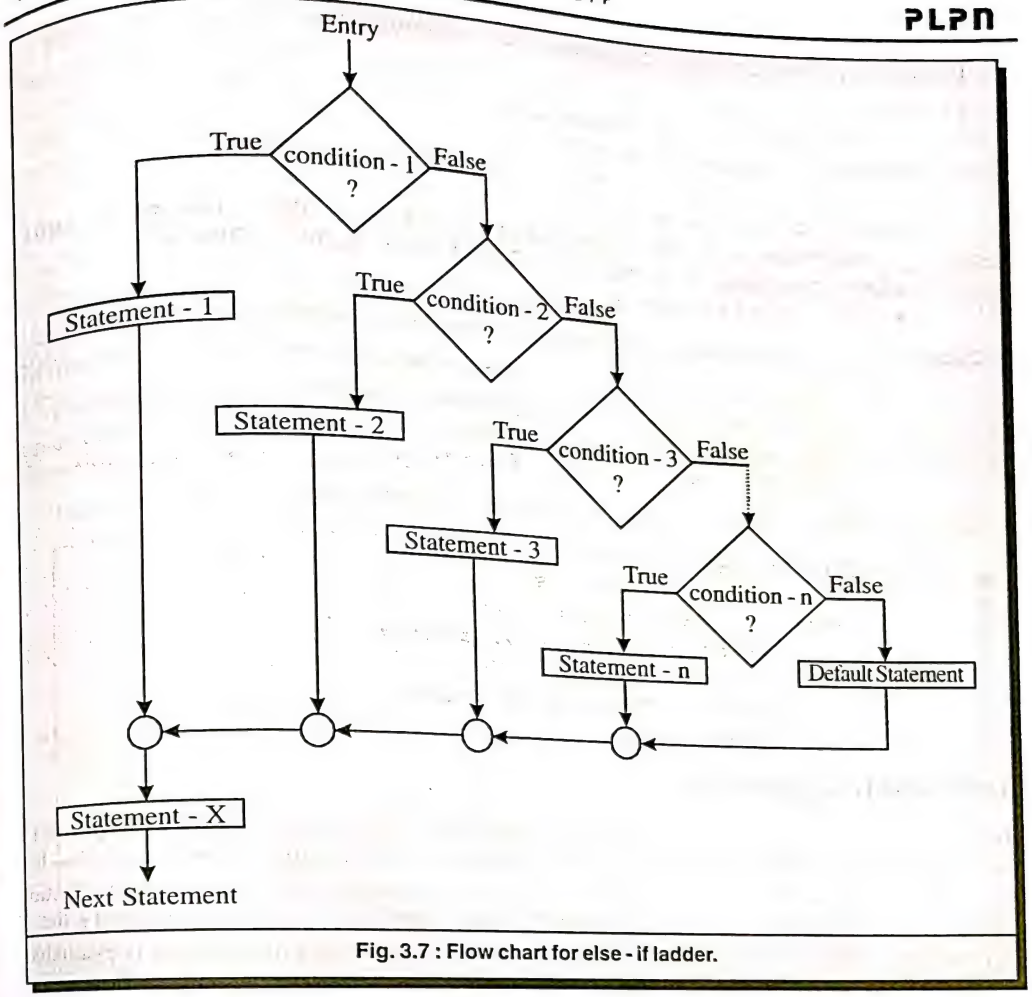


Fig. 3.7 : Flow chart for else - if ladder.

Let us consider some programming situations where you need to make use of else - if ladder.

1. The else if ladder is suitable in programs where we want to categorize based on conditions. Suppose we are computing electric bill based on units consumption.

up to 30 units	Rate Rs. 0.75 per unit.
31 to 100 units	Rate Rs. 2.50 per unit.
101 to 300 units	Rate Rs. 3.00 per unit.
Above to 100 units	Rate Rs. 4.60 per unit.

This policy can be implemented by else if ladder as shown below.

```
if (units < = 30)
     charges = 0.75 * units;
   else if (units < = 100)
     charges = 0.75 * 30 + 2.50 * (units - 30);
   else if (units < = 300)
     charges = 0.75 * 30 + 2.50 * 70 + 3.00 * (units - 100);
   else charges = 0.75*30 + 2.50*70 + 3.00 * 200 +
   4.60 * (units-300);
```

2. Suppose we want to allocate grading to students based on their percentage.

Percentage > 79	Grade is Honours.
Percentage > 59	Grade is first division.
Percentage > 49	Grade is second division.
Percentage < 49	Grade is fail.

This policy can be implemented by else - if ladder as shown below.

```
if (Percentage > 79)
        grade = "Honours" ;
   else if (percentage > 59)
        grade = "First division" ;
   else if (percentage > 49)
        grade = "Second division" ;
   else grade = "Fail" ;
```

The Switch Statement

When we need to select many alternatives, the program using if statement becomes complex one. The read ability of program decreases. Sometimes it may create confusions to the reader. C++ has built in multiway decision statement known as switch. It is also called as multiple-branch selection statement. The switch statement tests the value of a given variable (or expression) against a list of case values and when match is found, a block of statements associated with that case is evaluated. The general form of switch statement is

```
switch (expression)
  {
       case value - 1 :  block - 1
                         break ;
       case value - 2 :  block - 2
                         break ;
       ---------------
       ---------------
       default :         default block
                         break ;
  }
   statement - x ;
```


Here switch, case, default and break are keywords. The expression is an integer expression or character. That is value of the expression must evaluate to a character value or integer value. Floating - point or other expressions are not allowed. Value - 1, value -2, ------ are constants or constant expressions. They are known as case labels. Each of these values should be unique within a switch statement. Block - 1, Block - 2, ----- are statement lists and may contain zero or more statements. There is no need to but braces around these blocks. Case labels ends with colon.

When switch statement is executed, the value of the expression is successively compared against the values value - 1, value - 2, ------. If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signals the end of particular case and causes an exit from the switch statement, transferring the control to the statement - x.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and control goes to the statement - x.

The flow chart of switch statement is shown in figure 3.8.

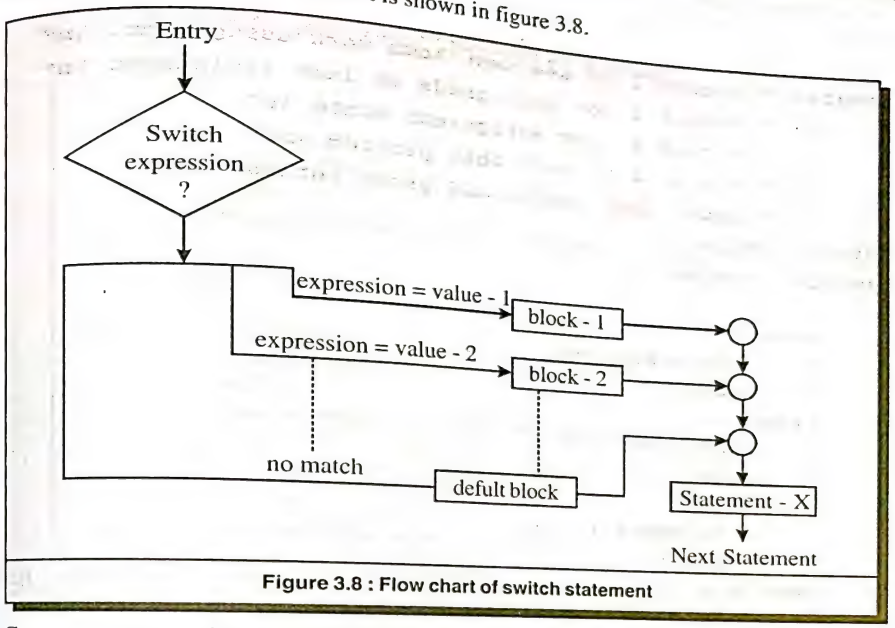


Figure 3.8 : Flow chart of switch statement

C++ can support at least 16,384 case statements. In practice, we would like to limit the number of case statements to a smaller amount for efficiency. The case statement can not exist by itself, outside of a switch.

There are certain important things to note about switch.

i) The switch statement differs from the if in that switch can only test for equality, whereas if can evaluate any type of relational or logical expression.

ii) No two case values can be identical. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.

iii) If character constants are used in the switch statement, they are automatically converted to integers.

iv) If you forget a break in a switch statement, the compiler will not issue an error message. The present case is executed and computer will go on to execute next case also. This may produce a puzzling output. When Computer starts executing a case, it does not stop unit it encounters either a break or end of the switch statement.

Now lets consider the programming situation where you need to use switch statement.

The switch statement is often used to process keyboard commands, such as menu selection. A menu in the restaurant presents a list of alternatives for a customer to choose from. A menu in computer program does the same thing ; it presents a list of alternatives on the screen for the users to choose from.

A program segment for menu of a program designed to give students information on home work assignment is shown.

```
int choice ;
cout<< " Choose 1 to see next home work assignment. \n"
    << " Choose 2 for your grade in last assignment \n"
    << " Choose 3  for assignment hints \n"
    << " Choose 4 to exit this program \n"
    << " Enter your choice and press return:" ;
cin >> choice ;
switch (choice)
 {
    case 1 :
         show_assignment ( ) ;
         break ;
    case 2 :
         show_grade ( ) ;
         break ;
    case 3 :
         give_hints ( ) ;
         break ;
    case 4 :
         cout << "End of Program \n" ;
         break ;
    default :
         cout << "Not a valid choice \n"
         << "choose again \n" ;
}
```

Observe that show_assignment ( ), show_grade ( ), give_hints ( ) are user defined functions.

Programming Examples

Problem 3.9

Write a program in C++ to find largest of three numbers.

Program

```
// 3.9 Largest of three numbers
#include <iostream.h>

void main()

{
    int a,b,c;

    cout<<"\n enter three integer numbers: ";
    cin>>a>>b>>c;
    cout<<"\na="<<a<<" b="<<b<<" c="<<c;

    if (a>b)
     {
         if(a>c)
         cout<<"\na="<<a<<" is largest\n";
         else
         cout<<"\nc="<<c<<" is largest\n";
       }
       else
       {
         if(b>c)
         cout<<"\nb="<<b<<" is largest\n";
         else
         cout<<"\nc="<<c<<" is largest\n";
       }

    }
```

Output Enter three integer numbers : 55 75 85

a = 55 b = 75 c = 85

c = 85 is largest

Problem 3.10

Write a program in C++ to fine roots of quadratic equation.

$ax^2 + bx + c = 0$

The roots are given as

$$x1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

$$x2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

There are three possible cases.

1. If $b^2-4ac < 0$ then roots are complex conjugate.
2. If $b^2-4ac = 0$ then roots are real and equal.
3. If $b^2-4ac > 0$ then roots are real and not equal.

Program

```
// 3.10 Roots of quadratic equation

# include<iostream.h>
# include<math.h>

void main()

  {
    float a,b,c,discrimt,x1,x2,temp;

    cout<<"\n enter a,b,c :";
    cin>>a>>b>>c;
    cout<<"\n a="<<a<<" b="<<b<<" c="<<c;

    discrimt=b*b-4.0*a*c;

    if(discrimt<0)
    {
       cout<<"\n complex conjugate roots\n";
    }

    else

    {
       if(discrimt==0)
{
    x1=-b/(2.0*a);
    cout<<"Repeted roots\n";
    cout<<"\n Real roots= "<<x1;
 }

 else
 {
    temp=sqrt(discrimt);
    x1=(-b+temp)/(2.0*a);
    x2=(-b-temp)/(2.0*a);
    cout<<"\n Real roots\n";
    cout<<"Real root-1= "<<x1;
    cout<<"\n Real root-2=  "<<x2;
 }

    }

  }
```

Output

```
Enter a, b, c : 1 3 1
Real roots
Real root – 1 = – 0.381966
Real root – 2 = – 2.618034
```

Problem **3.11**

An electric power distribution company charges its domestic customers as follows :

Consumption unit	Rate of charges
upto 30 units	Rs. 0.75 per unit
31 to 100 units	Rs. 2.50 per unit
101 to 300 units	Rs. 3.00 per unit
Above 301 units	Rs. 4.60 per unit

Write a program in C++ to read customer number and number of units consumed. Compute charges.

Program

```
// 3.11 Electric bill

#include<iostream.h>

void main()

{
  int units,custnum;
  float charges;

  cin>>custnum>>units;
  cout<<"\n customer number="<<custnum<<" Units="<<units;
```

```
    if(units<=30)
       charges=0.75*units;

       else if (units<=100)
         charges=0.75*30+2.50*(units-30);

         else if(units<=300)
        charges=0.75*30+2.50*70+3.00*(units-100);

       else charges=0.75*30+2.50*70+3.00*200+4.6*(units-300);
        cout<<"\n  charges=Rs."<<charges;

    }
```

Output

1234
100
customer number = 1234 units = 100
charges = Rs. 197.5

Problem 3.12

A company manufactures three products - engines, pumps and fans. They give a discount of 10% on the order for engines if the order is for Rs. 50,000 or more. The same discount of 10% is given on pumps order of Rs. 20,000 or more and on fans order of Rs. 10,000 or more. On all other orders they do not give any discount. Write a program in C++ which implements this policy.

Assume code 1: for engines
2: for pumps
3: for fans

A decision table for discount policy is

Product code	1	2	3	else
Order Amount	>= 50,000	>= 20,000	>= 10,000	
Discount	10%	10%	10%	0%

Program

```
// 3.12 Discount policy

#include <iostream.h>

void main()
{
int s_no,code;
float amt,discount,net_amt;

cin>>s_no>>code>>amt;

   switch(code)
    {

 case 1:
         if (amt >= 50000)
         discount=0.1*amt;
         break;

 case 2:
         if (amt >= 20000)
         discount=0.1*amt;
         break;

 case 3:
         if (amt >= 10000)
         discount=0.1*amt;
         break;

 default:
       discount=0.0;
       break;
  }

    net_amt=amt-discount;

   cout<<"\n S-no="<<s_no<<" code="<<code<<" discount="
    <<discount<<" net= "<<net_amt;


}
```


Output

10 1 50000
S- no = 10 code = 1 discount = 5000 net = 4500

Problem 3.13

Write a program to generate a magic number using standard random number generated rand(), which returns an arbitrary number. The program should print the message **Right** when the player guesses the magic number.

Program

```
// 3.13  Magic number

# include <iostream.h>
# include <stdlib.h>

void main()

{
    int magic;
    int guess;

    magic=rand();

    cout<<"\n Guess the magic number:";
    cin>>guess;

    if(guess==magic)
        cout<<"\n ** Right **";
     else cout<<"\n ** Wrong **";

}
```

Loops

Computer has ability to perform a set of actions repeatedly. This involves repeating a set of some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is called looping.

Looping can be achieved by using if statement. Apart from if, C++ provides three loop control structures. They are also called as interation statements. They are :

1. The while statement.
2. The do - while statement.
3. The for statement.

We will discuss these statements in following pages.

The while statement

The simplest in all looping structure in C++ is while statement. The general form of while is:

```
While (test condition)
    {
        body of loop
    }
```


Here while is a keyword. The test condition is an expression which may return zero or non zero value. A non zero value is treated as true value and zero vale as a false value. The body of loop is either an empty statement, a single statement or a block of statements.

The while is entry - controlled loop. The test condition is evaluated first and if the condition is true then the body of loop is executed. After execution of body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of body continues until the test condition finally becomes false and control is transferred out of the loop. On exit, the program continue with statement immediately after the body of the loop.

The test condition in while loop should become false at some time, otherwise the loop will be executed forever, indefinitely. If the condition is false at the beginning of while loop, then the loop is never executed.

Now lets consider some programming situations where you need to use while loop.

1. Suppose we want to obtain sum of first 100 natural numbers. We start with first number 1 and sum = 0. We go on adding number to sum and every time increment the number by one until we reach the number 100. This can be achieved by while loop as shown below.

```
sum = 0 ;
number = 1 ;
while (number < = 100)
  {
    sum = sum + numbers ;
    number = number + 1 ;
  }
```

   Here while loop will be executed until the condition number < = 100 is true. When number becomes 101, the condition returns a false value and control is transferred out of the loop to the next statement. The variable sum consist of sum of first 100 natural numbers.

2. Sometimes the loop is used as a delay loop and wait for user response. In such a case there need not be any statements in the body of while loop.

```
While (( ch = getchar ()) ! = 'A' ) ;
```

This statement will simply loop until the user types A. Here getchar () is built in function which is used to read character from keyboard.

The do - while Statement

In while loop, the condition is tested first before the loop is executed. If condition is false in first attempt then body of loop may not be executed even once. In some occasions it might be necessary to execute body of loop before test is performed. Such situations can be handled with do - while loop. The general form of do - while statement is :

```
do
    {
        body of loop
    }
while (test condition) ;
```

Here do - while are keywords. The test condition is an expression which may return true or false value. The body of loop is either an empty statement, a simple statement or a block of statements.

On reaching do statement, the program proceeds to evaluate the body of loop first. At the end of the loop, the test condition in while is evaluated. If the condition is true, the program continues to evaluate the body of loop once again. This process continues as long as condition is true. When condition becomes false, the loop will be terminated and control goes to the next statement after the while statement. In do - while the body of loop is executed at least once and such a loop is called exit controlled loop.

When body of loop consist of only one statement, then curvely braces are not necessary. We may write,

```
do
  statement ;
  while (test condition) ;
```

Now lets consider some programming situations where we need to use do - while loop.

1. Suppose we want to read the numbers from keyboard until it finds a number less than or equal to 100. We can write do - while loop for this purpose as follows.

```
do
  {
    cin >> number ;
  }
  while (number > 100) ;
```

2. The most common use of do - while loop is in menu selection function, because you will always want a menu function to execute at least once. After the options have been displayed, the program will loop until a valid option is selected.

The following program segment shows the use of do - while loop in menu selection.

```
char choice;
cout   << "1. Check Spelling\n"
       << "2. Correct Spelling Errors\n"
       << "3. Display Spelling Errors\n" ;
do
  {
      cin >> choice ;
      switch (choice)
```



```
  {
      case '1' : check_spelling ( ) ;
                 break ;
      case '2' : correct_errors ( ) ;
                 break ;
      case '3' : display_errors ( ) ;
                 break ;
  }
}while (choice !='1' && choice !='2' && choice !='3');
```

Here check_spelling ( ), correct_errors ( ), and display_errors ( ) are user defined functions.

The for - statement

The for loop is the most popular loop. It is widely used by the programmers. It is also an entry controlled loop. The general form of for statement is :

```
for (initialization ; test condition ; increment)
  {
      body of loop
  }
```

Here for is a keyword. Body of loop is either an empty statement, a single statement or a block of statements. The initialization is an assignment statement that is used to set the loop control variable. The test condition is relational expression that determines when the loop exits. The increment defines how the loop control variable changes each time the loop is repeated. The initialization, test condition and increment are seperated by semicolon. The execution of for statement is as follows.

1. Initialization of control variable is done first, using assignment statement such as i = 1 and count = 0. The variables i and count are known as loop control variables.
2. The value of the control variable is tested using test - condition. The test condition is relational expression such as i < = 10 that determines when the loop will exit. If the condition is true, the body of loop is executed otherwise the loop is terminated.
3. After the execution of body of loop, the control is transferred back to the for statement. Now the control variable is again tested to see whether it satisfies the loop condition. If condition is satisfied, the body of loop is executed again. This process continues till the value of control variable fails to satisfy the test condition.

In 'for' loop, the condition is tested at the beginning, hence body of loop may not be executed even at once if condition fails at start.

If necessary, we may omit more than one section of for loop. We may write

```
for ( ; m ! = 10 ;)
```

Here initialization and increment section is omitted. Observe that semicolons are needed in parenthesis.

Now lets consider some programming situations where you will need to use for loop.

1. Suppose we want to add 1 to 100 natural numbers. A program segment using for loop is shown below.

```
sum = 0 ;
for (x = 1 ; x < 100 ; x + +)
  {
      sum = sum + x ;
  }
```

Here x is control variable. X is initialized to 1 and then compared with 100. Since x is less than 100, x is added in sum. x + + causes the value of x to be increased by 1 and again tested to see if it is still less than or equal to 100. If it is, x is added to sum. This process repeats until x is greater than 100, at which point the loop terminate. The variable sum consist of sum of 1 to 100 natural numbers.

2. 'For' loop is more powerful, flexible and applicable. It is possible in the for loop to use more than one control variable to control the loop. This can be accomplished by using comma operator.

Suppose we have two variables x and y to control the for loop. They can be initialized inside the for statement as :

```
for (x = 0, y = 0; x + y < 10 ; + + x)
  {
    --------
    --------
  }
```

Comma seperates the two initialization statement.

3. We can set up time delay loop using null statement.

for (j = 1000 ; j > 0 ; j = j – 1) ;

This loop will be executed 1000 times without producing any output. It simply causes time delay.

4. If we write ,

for (; ;) cout << "Infinite loop" ;

Then the loop will be executed infinitely. When conditional expression is absent, it is assumed to be true.

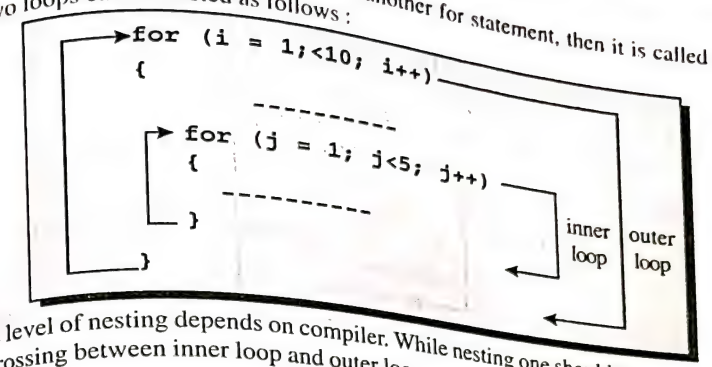
We can write a loop to get user response.

```
for ( ; ; )
{
  ch = greater ( ) ;
  break ;
}
```

This loop will run until user types any character at keyboard. Here break statement inside body of loop causes immediately termination. We are going to discuss break in detail later in this chapter.

Nesting of for loop

When one for statement is used within another for statement, then it is called nesting of for loops. Two loops can be nested as follows :



The level of nesting depends on compiler. While nesting one should take care that there should not be crossing between inner loop and outer loop.

The nested loops are required when we want to read data elements of a table. A table consist of rows and columns. One loop is formed for row and the other is formed for column.

While working with arrays, we often require nested loops.

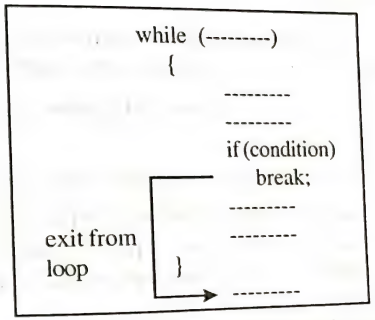
Break Statement

Loops executes certain set of statements repetedly until the control variable fails to satisfy the test condition. The number of times the loop is repeated is decided in advance and test condition is written to achieve this. Some times, when executing a loop it becomes desirable to skip a part of the loop or to leave loop as soon as certain condition occurs. Suppose we are searching for a name in 100 names. A program loop will be written to get executed 100 times, but as soon as we find desired name, we want the loop to get terminated.

An early exit from the loop can be accomplished by using break statement. When break statement is encountered in side a loop, the loop is immediately exited and entire program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, break will exit only a single loop.

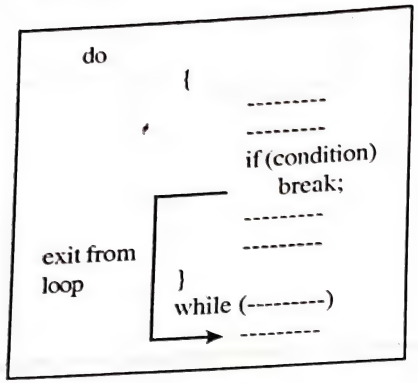
The break statement can be used with while loop, do - while loop and for loop.

Consider a while loop :



Within body of while loop we have used break. The break is normally used with if statement. If condition becomes true, the control is transferred out of the loop.

The use of break in do - while loop is shown below.



Observe that in every loop, break is used with an if statement.

When break is used within nested loop then it causes an exit form only the innermost loop.

for example,

```
for (t = 0 ; t < 100 ; ++t)
 {
    cout = 1 ;
    for (; ;)
      {
        cout << count ;
        count +++ ;
      if (count = = 10) break ;
      }
 }
```

The above program segment prints the numbers 1 through 100 on the screen 100 times. Each time execution encounters break, control is passed back to the outer for loop.

A break used in the switch statement will affect only that switch. It does not affect any loop the switch happens to be in.

Now lets consider some programming situations where break statement is used.

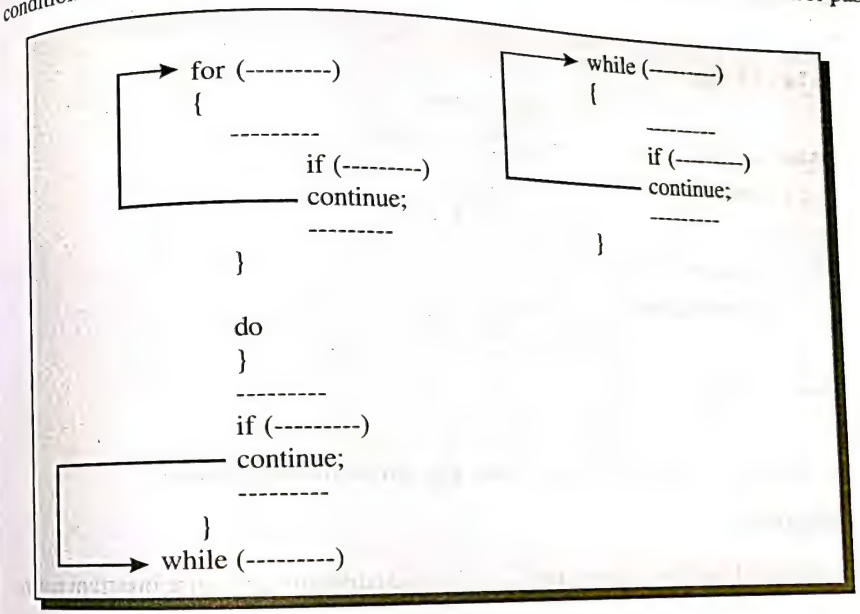
1. The break statement can be used to display error messages. Suppose we are writing program to add negative numbers. If a zero or positive number is entered, we do not want to add this number, instead we will display error message. The following program segment shows use of break for this situation.


```
while (++ count < = 10)
{
   cin >> number
   if (number > = 0)
   {
     cout << "ERROR : zero or positive number"
       << "was entered \n" ;
     break ;
   }
   sum = sum + number ;
}
```

Continue statement

Sometimes we want to skip statements inside the loop and want to take control to the beginning of the loop. The continue statement allows us to do this.

When continue is used inside any loop, control automatically passes to the beginning of the loop. When we use continue within for loop, continue causes the conditional test and increment portions of the loop to execute. When it is used in while and do - while loops, program control passes to the conditional tests as shown below :



Observe that continue statement is also used with an if statement.

Here remember that break causes loop to be terminated while continue, causes the loop to be continued with next iteration after skipping certain statements. Continue tells the compiler :

"Skip the following statements and continue with next iteration".

Now lets consider some programming situations where we can use continue.

1. Suppose we are writing program to obtain reciprocal of given integer number. We can make a provision to prevent division by zero by using continue as shown below :

```
for (i = 0 ; i < 4 ; i + +)
  {
    cin >> number ;
    if (number = =  0)
     continue ;
    k = 1.0 / number ;
    cout << k ;
  }
```

2. Continue can be used to exit from the loop. Suppose you are writing a program which codes a message by shifting all characters you type one letter higher. For example, A becomes B. The termination of program can be achieved by entering $. The exit from the loop can be achieved by using continue as shown below :

```
char ch, done ;
done = 0 ;
while (! done)
{
  cin >> ch ;
   if (ch = =  '$')
    {
         done = 1 ;
         continue ;
    }
  cout << (ch + 1) ;
}
```

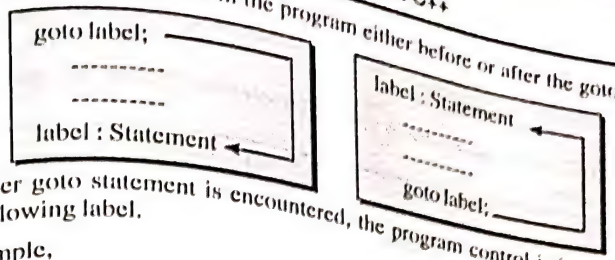
When a $ is entered, the variable done will be true and causes the loop to exit.

Labels and goto

Goto is unconditional branching statement of C++. A goto statement can cause program control to end up any where in the program checking for any condition.

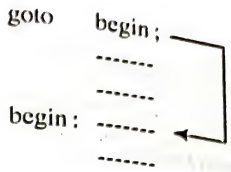
A goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by colon. A label is placed immediately before the statement where the control is to be transferred. The general form of goto and label are shown below.

The label can be any where in the program either before or after the goto statement.



Whenever goto statement is encountered, the program control is immediately transferred to statement following label.

For example,

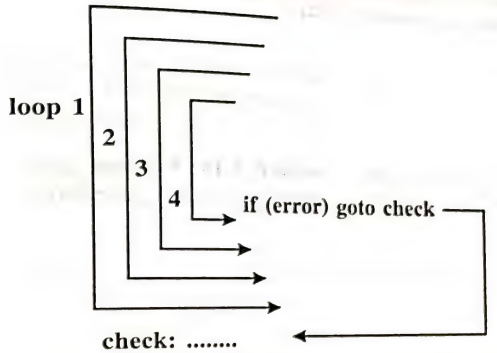


Now lets consider some programming situations where you can use goto.

1. The goto statement can be used to form a loop. The following program segment shows a loop of 1 to 100 using a label and goto.

```
x = 1 ;
loop1 :
    x ++ ;
    if (x < 100) goto loop1 ;
```

2. Another use of goto is that we can come out of a deeply nested loops when an error occurs, as shown below.



Goto statements breaks the normal sequential execution of the program. With goto we can never be sure how we got to a certain point in the program. A good programmer always avoids the use of goto. The other statements like if, for, switch, while are more logical and hence they can be used in place of goto. When goto is used, many compilers generate less efficient code, the programming logic becomes complicated and programs are unreadable with goto.

Programming Examples

Problem 3.14

Write a program in C++ to read tender-id and tender amount. Pick up the maximum tender among the group of tenders.

Program

```
//  3.14 Maximum tender

#include<iostream.h>

void main()

{
  int tender_id,max_id,n,i;
  float tender_amt,max_tender=0;

  cout<<"\n enter no of tenders \n";
  cin>>n;

  for (i=1;i<=n;i++)
  {
    cin>>tender_id>>tender_amt;

    if(tender_amt>max_tender)
    {
      max_tender=tender_amt;
      max_id=tender_id;

    }

  }

 cout<<"\n maximum tender-id= "<<max_id;
 cout<<"\n maximum amount="<<max_tender;

}
```

Output

Enter no. of tenders

2 101 500 102 600

maximum tender - id = 102

maximum amount = 600

Problem 3.15

Write a program in C++ to find sum of the following series :

$$sum = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \text{------} \frac{(-1)^n x^{2n-1}}{(2n-1)!}$$

The i^{th} term of series $= \frac{(-1)^{i-1} x^{2i-1}}{(2i-1)!}$

The $i-1^{th}$ term $= \frac{(-1)^{i-2} x^{2i-3}}{(2i-3)!}$

$\therefore$ i^{th} term $= \{ (-1) x^2 / (2i-2)(2i-1) \} * (i-1)^{th}$ term.

Program

```
// 3.15  Sum of series

#include<iostream.h>

void main()

{
  int i,n,denominator;
  float x,sum,term;

  cin>>x>>n;

  for(sum=x,term=x,i=2;i<=n;++i)

  {
     denominator=(2*i-2)*(2*i-1);
     term*=(-x*x)/denominator;
     sum+=term;
  }

   cout<<"\n sum="<<sum<<" x="<<x<<" n="<<n;

}
```

Output

0.1 5

sum = 0.099833 x = 0.1 n = 5

Problem 3.16

Give a set of point (x_1, y_1), (x_2, y_2) ----- (x_n, y_n), it is required to fit a straight line.

$$y = mx + c$$

Through these points which is the best approximation to these points, i.e. optional values of m and c in the above equation for the straight line are to be found.

$$m = \frac{n\Sigma x_i y_i - \Sigma x_i \Sigma y_i}{n\Sigma x_i^2 - (\Sigma xi)^2}$$

and $c = (\Sigma y_i - m\Sigma x_i)/n$.

Σ is summation for $i = 1$ to n.

Write a program to read n pairs of values (x , y) and compute m and C.

Program

```
// 3.16  Fitting of straight line

#include<iostream.h>

void main()

 {
    float
sum_x=0,sum_y=0,sum_xy=0,sum_xsq=0,x,y,numerator,d,m,c;
    int i,n;

    cin>>n;

    for(i=1;i<=n;++i)
    {
      cin>>x>>y;
      sum_x+=x;
      sum_y+=y;
      sum_xy+=x*y;
      sum_xsq+=x*x;
    }

     numerator=n*sum_xy-sum_x*sum_y;
     d=n*sum_xsq-sum_x*sum_x;
    m=numerator/d;
    c=(sum_y-m*sum_x)/n;

    cout<<"\n slope m="<<m<<" intercept c="<<c;

}
```


Output

```
5
12  13  45  76  23  56  56 78  56 88
slope m = 1.38843 intercept c = 8.884297
```

Problem 3.17

A set of integers is given. Write a program to count number of positive integers, negative integers and zero in set.

Program

```
// 3.17  To count negative,positive and zero
#include<iostream.h>

void main()

{
   int no,no_p=0,no_zero=0,no_n=0,i=1;

  do
  {
     cin>>no;
     if (no>0)
       ++no_p;

       else if(no==0)
       ++no_zero;

   else ++no_n;
   i++;
    }
    while(i<=10);

       cout<<"\n no. of positive integers="<<no_p
         <<" no. of negative integers="<<no_n
         <<" no. of zero integers="<<no_zero;

}
```

Output

```
11  21  31  –40  0  45  23  34  676  – 67
number of positive integers = 7 number of negative integers = 2 no of zero integers = 1.
```

Problem 3.18

A class of n students take an annual examination in m subject. Write a program to read marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them.

Program

```
// 3.18 To compute total marks

#include<iostream.h>

void main()

{

 int n,m,i,j,no,marks,total;

 cout<<"\n enter no of students & subjects\n";
 cin>>n>>m;

 for(i=1;i<=n;++i)
 {
    cout<<"\n enter roll no\n";
    cin>>no;

    total=0;
    cout<<"\n enter marks of "<<m<<" subjects for
    "<<no<<":";

    for (j=1;j<=m;j++)
    {
    cin>>marks;
    total=total+marks;
    }

    cout<<"\n total marks="<<total;

 }

}
```

Output

```
enter no of students & subjects
2      3
enter roll no
101
enter marks of 3 subjects 101 :  67  87  78
total marks = 232
```



```
enter marks of 3 subjects 102 :  45  56  78
total marks = 179
```

Problem 3.19

Write a program to compute x to the power n. (x^n)

Program

```
//   3.19   Computation of x to the power n

#include<iostream.h>

void main()

{
    int count,n;
    float x,y;

    cout<<"\n enter x & n:";
    cin>>x>>n;

    y=1.0;
    count=1;

    while(count<=n)
    {
      y=y*x;
      count++;
    }

    cout<<"\n x="<<x<<" n="<<n<<" X to the power n="<<y;

}
```

Output

```
enter x & n : 0.6 5
x = 0.6        n = 5     x to the power n = 0.07776
```

Problem 3.20

Write a program to convert binary number to decimal number. If input binary is (1111) then its decimal equivalent is (15).

Program

```
//   3.20   Conversion of binary number to decimal number

#include<iostream.h>

void main()
```

```
{
   int bin,binary,decimal=0,digit,base=0;

   cout<<"\n input binary number:";
   cin>>binary;
   cout<<"\n"<<binary;

   bin=binary;

   while(binary)
   {
     digit=binary%10;
     decimal+=digit<<base;
     base+=1;
     binary/=10;
   }

   cout<<"\n Decimal equivalent of binary num   =
   "<<decimal;

}
```

Output

Input binary number : 1111

1111

Decimal equivalent of binary number = 15.

Problem 3.21

Write a program to compute cosine series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \text{-----} \frac{x^n}{n!}$$

Make provision to enter x in degrees.

Program

```
// 3.21 Cosine series

#include<iostream.h>
#include<math.h>

void main()

{
 float x,t,sum;
 int i,n=20;
```



```
 cout<<"\n input x:";
 cin>>x;
 cout<<"\n"<<x;

 /* converting x into radians */
 x=x*3.142/180;
 t=1;

 sum=1;

 for(i=1;i<n+1;i++)
 {
    t=t*pow((double)(-1),(double)(2*i-1))*x*x/
    (2*i*(2*i-1));
    sum=sum+t;
 }

 cout<<"\n cos(x)="<<sum;

}
```

Output

input x : 45

45

cos (x) = 0.707035

Problem **3.22**

Write a program to compute exponential series.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \text{-----} + \frac{x^n}{n!}$$

Program

```
//   3.22   Exponential series

#include <iostream.h>

void main()

{

  float x,t,sum;
  int i,n,prod;

  cout<<"\n Input x & n:";
```

```
    cin>>x>>n;


    t=1;
    sum=1;
    for(i=1;i<n;i++)
    {
     prod=i;
     t=t*x/prod;
     sum=sum+t;
    }

    cout<<"\n e raised to power x = "<<sum;

 }/* End of main */
```

Output

Input x & n : 5 8

e raised to power x = 1.648721

Problem 3.23

Write a program to construct pyramid of digits.

Program

```
// 3.23 Pyramid of digits

#include<iostream.h>

void main()

{

  int p,m,q,n;

  cout<<"\n Enter number of lines:";
  cin>>n;
  cout<<"\n\n";

  /* To print spaces */

  for(p=1;p<=n;p++)

  {
```



```
    for(q=1;q<=n-p;q++)
     cout<<"   ";

    /* To print digits */
    m=p;
    for(q=1;q<=p;q++)
     cout<<"   "<<m++;
     m=m-2;

    for(q=1;q<p;q++)
     cout<<"   "<<m--;
    cout<<"\n\n";

  }
  cout<<"\n";

 }
```

Output

Enter number of lines : 4

```
      1
    2 3 2
  3 4 5 4 3
4 5 6 7 6 5 4
```

Problem 3.24

Three positive integers a, b and c, such that a < b < c, form a phythagorean triplet if $a^2 + b^2 = c^2$.

Write a program that generates all phythagorean triplets a, b, c where a, b ≤ 25.

Program

```
// 3.24  Phythagorean triplet

#include <iostream.h>
#include <math.h>

#define LIMIT  25

void main()

{

  int a,b,c,c_sqr;
```

```
    for(a=1;a<LIMIT;a++)

      for(b=a+1;b<=LIMIT;b++)

    {
      c_sqr=a*a+b*b;
      c=sqrt(c_sqr); /* truncate fraction */

      if(c*c==c_sqr)
      cout<<"\n"<<a<<" "<<b<<" "<<c;

    }
}
```

Output

3	4	5
5	12	13
6	8	10
7	24	25
8	15	17
9	12	15
10	24	26
12	16	20
15	20	25
18	24	30
20	21	29

QUESTIONS - 3.2

1. What is statement in C++ ?
2. What is block of statements ?
3. What are control structures in C++ ?
4. Along with Syntax explain if - else statement.
5. Explain the use of else - if ladder.
6. Explain the switch statement of C++.
7. What are loops ?
8. Compare between while and do - while statements.
9. What is Syntax of for loop ?
10. How will you achieve an early exit from loop ?
11. Explain the purpose of continue statement.
12. What is goto statement ? Why should we avoid use of goto ?
13. Differentiate between -
    i) break and goto
    ii) break and continue.
14. Write a program to read decimal number and display its binary form.
    for example, if input number is 2, the
    binary equivalent of 2 is 0010.
15. Write a program to read name, age, sex of 100 persons in an organisation. Determine average age of males and females in the organisation.
16. Write a program to read hexadecimal number and display its decimal equivalent.
    for example, $(A2)_{16} = A\,(10) \times 16^1 + 2 \times 16^0$
    $= 160 + 2 = (162)_{10}$
17. The equation $x^2 + y^2 = r^2$ represents a circle with center at the origin and radius r. Write a program that reads r and determines the number of points with integer coordinates that lie within the circle.
18. Write a program that, for all positive integers i, j, k and l from 1 through 1000, determines all combinations i, j, k and l such that $i + j + k = l$ and $i < j < k < l$.
19. Write a program to find sum of following series.
    i) sum $= 1^2 + 2^2 + 3^2 + \text{-----} + n^2$
    i) sum $= 1^3 - 2^3 + 3^3 - \text{-----} + n^3$
    i) sum $= 1 - 3^2 + 5^2 - \text{-----} + n^2$
20. Write a program to read a positive integer and generate numbers in following form.
    Enter number : 5
    Output : 5 4 3 2 1 0 1 2 3 4 5

Functions

Many times we hire the services of other people in order to get a job done. Any complex job can not be performed by a single person. Consider a familiar example of marriage ceremony. This is big event and so many persons are involved in. The food contract is given to one, the other arrangements are looked after by other. We hire a hall for ceremony. For such a event we need to make planning. We need to divide job into small jobs and get it assigned to individuals. Even in our day to day regular activities you will find that we need to hire services. Your bike is maintained by mechanic and your garden is maintained by a gardener. Similarly while attempting to solve complex job by computer we find one program can not handle all tasks by itself. It requires other subprograms to get the task done. These subprograms are called as functions. Functions are building blocks of C++ programs. In C++ many new features are added to functions. In this section we will study these functions.

What is a function

A function is a self contained block of statements that performs a particular task. They may be developed seperately and tested. A name is given to each function and we can specify the method of sending data to it. The functions may be linked together.

In C++, functions can be classified into two categories.

i) Library functions
ii) User defined functions

Library functions are predefined functions. These functions are not required to be written by us. The code for these functions is readily available. You can use these functions in your program. When we use these functions in program, we need to include corrosponding header file (with .h extension) in the program, in which the code for functions is defined. Table 3.14 shows some library functions along with the header file.

Name	Description	header file
sqrt	square root	math. h
pow	powers	math. h
abs	absolute value for int	stdlib. h
ceil	ceiling (round up)	math. h
floor	floor (round down)	math. h

Table 3.14 Some Library functions

User defined functions are developed by programmer. These are developed as per the need of the program. After development it can be a part of C++ program library. The main ( ) is an example of user defined functions.

Advantages of functions

Functions are building blocks. Every C++ program can be thought of as collection of functions. The program execution starts from main function. All of the code for program can not be incorporated into main because it leads to number of problems. The program may become too large and complex and hence debugging, testing and maintenance becomes difficult. If the program is divided into functional parts, then each part may be independently coded and later combined into single unit.

The advantages of functions are :

i) Easy to write correct small functions for small tasks.
ii) Easy to read, write and debug a function.
iii) The maintenance and modification of functions is easier.
iv) The length of source program can be reduced by using functions at appropriate places.
v) A function once developed, may be used by many other programs so it avoids rewrite the same code again and again.

Function Prototyping

In C++ all functions must be declared before they are used in program. This is achieved by function prototyping. The prototype is a declaration that defines the arguments passed to the function and the type of the value returned by the function. It has the form :

type function_ name (type arg1, type arg2 ,..... type argn) ;

Here type specifies type of value returned by function. Function_name is any valid variable name other than keyword. The arg1, arg2, ----- argn are n arguments. Before names of arguments, the type of argument is mentioned. The use of arguments names is optional however writing their type is mandatory. Function prototype appears before the function is called. Normally function prototypes are written before main function. The function prototype tells compiler everything that it need to know about function. It tells name of function, number of arguments the function needs, and the type of arguments.

For example, we may write function prototype as :

double total_cost (int number_per, double price_par) ;

Here double specifies the type of value returned by function total_cost. The argument names number_par and price_par are called as formal parameters or formal arguments. The data type of number_par is int and that of price_par is double. Observe that function prototype ends with semi colon.

Defining a function

A function definition describes how the function computes the value it returns. The function definition consist of a function header written as the same way as the function prototype, except that the header does not have a semicolon at end.

The function body consist of declarations and executable statements enclosed within a pair of braces. The function body is similar to that of main ( ) part of program. When a function is called in main ( ), the values to formal parameters are passed and then the statements in body of function are executed. The last statement in body is return statement. It consist of a keyword return followed by

an expression. If determines value returned by function.

The general form of function definition is :

```
type function _ name (type arg1,----- type argn)
{
    Local variable declarations ;
    executable statement 1 ;
    executable statement 2 ;
    ----------
    ----------
    return (expression) ;
}
```

The variables that are defined within a function are called local variables. They are valid only within function and they are destroyed on exit from function. A function's code is private to that function and cannot be accessed by any statement except through a call to function. i.e. you can not use goto to jump into the middle of the function.

For example. we may write a function definition for total - cost as shown below :

```
double total _ cost (int number _ par, double price_par)
{
    const double TAX_RATE = 0.05 ;
    double subtotal ;
    subtotal = price_par * number_par ;
    return (subtotal + subtotal * TAX_RATE) ;
}
```

function body

Observe that in above function definition, number_par and price_par are formal parameters. They will be passed values when function total_cost is called in main ( ). subtotal is local variable. The return statement returns a double value to the calling function.

Call by value

A function can be called in main ( ) by writing the name of the function.

For example, we may write statement

```
bill = total_cost (number, price) ;
```

Here total_cost is function name and number, price are variables known as actual parameters. The values of number and price are passed to the formal parameters, that is the values of number and price are copied to formal parameters, number_par and price_par. This process of passing values is known as call by value. In call by value, the value of first actual parameter is passed to first formal parameter and the value of second actual parameter is passed to second formal parameter and so on depending on number of parameters. For call by value, the type, order and number of actual and formal parameters must be same. After passing values, the function body is executed and return value is assigned to variable bill. During execution of function body it may be possible that the values of formal parameters may get changed but it will not have any effect on values of actual parameters.

Lets consider another example where a function sqr computes square value as shown below.

```
int sqr (int x) ;          <-- function prototype
main ( )
{
    int t =  10 ;
    m = sqr (t) ;          <-- function sqr is called (call by value)
}
int sqr (int x)
{
    x =  x * x ;
    return (x) ;
}
```

function definition for sqr

Apart from call by value, there is another method by which values of arguments can be passed to function. This is known as call by reference.

We will discuss call by reference lateron in the chapter on pointers.

The return statement

When a function is called, the control of execution gets transferred to function. There are two ways that a function terminates execution and returns to the caller. The first occurs when the last statement in the function has executed i.e. when closing brace } is encountered. This is the default method of terminating the execution of function.

In second method, we can use return statement to stop execution. A return statement may send value to the calling function. A return statement can return only one value per call at the most.

The return statement can have several forms.

For example we may have a plane return as :

```
return ;
```

The plane return does not return any value but it only returns the control of execution to the calling function.

```
return (expression) ;
```

This form of return, returns values of expression. Also, the return need not always be present at the end of called function.

For example,

```
fun ()
{
    char ch ;
    cout << " Enter any alphabet " ;
    cin >> ch ;
    if (ch > = 65 && ch < = 90)
        return (ch) ;
    else
        return (ch + 32) ;
}
```

In this function different return statements will be executed depending on whether ch is capital or not.

Returning values

All functions, except those of type void, return a value. The value is specified by the return statement. In C++, a non void function must contain a return statement that returns a value. If the return statement is absent then any garbage value is returned. The absence of return statement is not a syntax error, still care should be taken to avoid it.

If the function is not declared as void, you may use it in an expression.
For example,

```
x = power (y) ;

if (max (x, y) > 100) cout << "greater" ;
```

Observe that in above expressions, power (y) and max (x, y) are functions.

As a general rule, a function can not be the target of an assignment. A statement such as

```
max (x, y) = 100 ;
```

is wrong. The C++ compiler will give an error.

When you write programs, your functions generally will be of three types. The first type is simply computational. These functions are specially designed to perform operations on their arguments and return a value based on that operation.

A computational function is a pure function. The standard library functions such as sqrt ( ) sin ( ) are examples of computational functions.

The second type of function manipulates information and returns a value that simply indicates a success or failure of that manipulation. An example is the library function fclose ( ), which is used to close a file. If the close operation is successful it returns 0 ; if the operation is unsuccessful, it returns EOF.

The last type of function has no explicit return value. This function is strictly procedural and produces no value. An example is exit ( ), which terminates a program. The functions which are not returning any value should be declared as returning type void. By declaring a function void, you keep it from being used in an expression, thus prevents accidental misuse.

If the function is returning some value and if the value is not assigned to any variable, in such a case the returned value is simply discarded.

The main ( ) function returns an integer to the calling process, which is generally the operating system. If main () does not return a values explicitly ; most of C++ compliers automatically return 0.

Scope rules

When we write a program, we need to make use of different variables. Some variables are declared and used within certain function, while some other variables are declared outside function and are used in different functions. All the variables are not valid in all the functions. To understand which variables are valid over what part of the program we need to study scope rules.

The scope of the variable determines over what part of the program a variable is actually available for use. The scope of a variable depends on storage class of variable.

In C++, there are four storage classes.

1. Local or internal variables.
2. External variables.
3. Static variables.
4. Register variables.

We will discuss each of these variables.

Local or internal variables

Local or internal variables are declared inside a function in which they are to be utilized. They are created when a function is called and destroyed automatically when the function is exited. Hence some times they are called automatic variables.

A variable declared inside a function without a storage class specification is, by default, a local variable. We may declare and use the same variable name in different functions of same program.

For example,

```
void  display (int i) ;
main ( )
{
      int i = 20 ;
      display (i) ;
}
void display (int j)
{
      int k = 35 ;
      cout << j ;
      cout << k ;
}
```

In this example, the presence of i is known apply to function main ( ) and not to any other function. Similarly the variable k is local to function display ( ).

The scope of local variable is local to the function in which it is defined.

External variables or global variables

Variables that are active and alive throughout the entire program are called as external variables or global variables. These variables can be accessed by any function in the program.

A large program in C++ may consist of different modules which can be seperately complied and linked together. For this, we require some way of telling all the files about the global variables required by the program. In C++, you can define the global variable only once.

That is in a multifile program you can define all the global variables in one file and use keyword extern for declaration of those variables in other files. The keyword extern tells the compiler that the variables types and name that follows is defined some where else.

In a single file program you can declare a global variable before main ( ). This global variable you can access within a function with keyword extern or even without keyword extern. If the compiler finds a variable that has not been declared within the current block, the compiler checks it matches any of the variables declared within enclosing blocks. If it does not then compiler checks the global variables. If a match is found, the compiler assumes that the global variable is being referenced.

For example,

```
int length ;          <-- Global declaration
main ( )
{
extern int length ;   <-- use of extern declaration
  ----------
  ----------
}
```

It is also possible to define variable after the function.

For example,

```
func1 ( )
{-----
      extern int y ;
} -----
int y ;
```

Here external declaration of y inside function tells compiler that y is defined some where else in program. Note that extern declaration does not allocate storage space for variables.

Static variables

The value of static variables persists until the end of the program. A variable can be declared static by using keyword static.

The static variable may be either internal or external.

When you apply the static modifier to a local variable, the compiler creates a permanent storage for it, as it creates for global variable. But the key difference between static local variable and global variable is that the static local variable remains known only to the block in which it is declared. A static local variable is a local variable that retains its value between function calls. We can use the static variable to count the number of calls made to a function.

A static variable is initialized only once, when the program is complied. It is never initialized again.

Static local variables are important to the creation of stand alone functions because several times we need to preserve the value between calls. If static local variables were not allowed, we may be needed to use global variables, but they may have side effects.

Now lets consider some programming situations where we need to use static local variables.

1. Suppose we are writing a function for number series generator that produces a new value based on the previous one. In such a case we can use a variable declared as static local variable as shown below.


```
int series ()
{
   static int series_num ;
   series_num = series_num + 25 ;
   return series_num ;
}
```

In this example, the variable series_num stays in existence between function calls. i.e. each call to series ( ) will produce new number in series based on preceding number without declaring that variable globally.

2. A static variable can be initialized. The value is assigned only once, at program - start - up and not each time the block of code is entered, as with normal local variable.

For example, consider the program :

```
int stat ( ) ;
main ( )
{
  int i ;
  for (i = 1 ; i < = 3 ; i + +)
  y = stat ( ) ;
}
int stat ( )
{
  stat int x = 0 ;
  x = x + i ;
  cout << x << endl ;
  return x ;
}
```

In above program during first call to stat ( ). , x is increment to 1. Because x is static this value persists and therefore, the next call adds another 1 to x giving it a value 2. The value of become 3 when third call is made.

The output of above program is :

```
x = 1
x = 2
x = 3
```

If we declare x as only local variable in place of static, then the output would have been :

```
x = 1
x = 1
x = 1
```

Static global variable

When we apply static specifier to a global variable then it instructs the compiler to create a global variable that is known only to the file in which you declare it. i.e. even though variable is global, functions in other files may have no knowledge of it.

Register variables

We can tell the compiler that a variable should be kept in machine's registers, instead of keeping in memory. The register access is much faster than a memory access. The variables or objects which are frequently used are kept in registers which leads to faster execution of program. This is done as follows :

```
register int count ;
```

Characters and integers can be stored in registers. Large objects such as arrays can not be stored in registers.

You can store any number of variables in registers. But only those are stored, which are frequently referenced. If many of the variables are declared as register variables, the compiler will automatically transforms register variables into non register variables, when the limit is reached. The limit depends on execution environment.

For example, suppose we are writing program for sum of m natural numbers. In this program there are certain variables which are referenced more and hence they can be declared as register variables as shown below.

```
int sum (register int m)
 {
    register int i ;
    register result = 0 ;
    for (i = 1 ; i < = m ; i ++)
    result + = i ;
    return (result) ;
 }
```

In this example, i, m and result are declared as register variables because they are all used within the loop.

Programming Examples

Problem 3.25

Write a function in C++ which reverses digits of an integer. Call function in main ( ).

Program

```
 //  3.25  Reversion of digits

# include<iostream.h>

int reverse(int no);

void main()

 {
   int p,q;
```



```
   cout<<"enter number:";
   cin>>p;

   q=reverse(p);

   cout<<"\n reverse of "<<p<<" is "<<q;

}/*end of main*/

int reverse(int n)

{

  int digit,rev_n=0;

  while(n!=0)
  {
     digit=n%10;
     rev_n=rev_n*10+digit;
     n=n/10;
   }

   return(rev_n);

  }
```

Output

Enter number : 1234
reverse of 1234 is 4321

Problem 3.26

The frictional force acting on particle is given by

$$f(y) = \begin{cases} 0 & \text{for } |y| \le 0.5 \\ 2y^2 & \text{for } |y| > 0.5 \end{cases}$$

Write a program in C++ to use this function in computing the value of

$$g = (mv/t) - f(v^2) + at$$

Program

```
 //   3.26 Frictional force and computation of g

 # include<iostream.h>

 float friction(float a);
 float abs(float p);
```

```
void main()

{

  float g,m,v,a,t;

  cout<<"\n enter m,v,a,t:";
  cin>>m>>v>>a>>t;

  g=(m*v/t)-friction(v*v)+a*t;

  cout<<"\n g="<<g;

 }  /* end of main*/


float friction(float y)

 {

   if(abs(y)<=0.5)
     return(0.0);

     else return(2.0*y*y);

  }

float abs(float x)

   {
     if(x<0)
       x=-x;

       return(x);

    }
```

Output

enter m, v, a, t : 100 2 6 10
g = 48

Problem 3.27

Write a function in C++ to compute factorial of a number by using formula

n ! = 1 × 2 × 3 ----- × n

Program

```
 //   3.27 Factorial of number
# include<iostream.h>

long int factorial(int a);

void main()

{
  int b;
  long int fact;

  cout<<"\n enter number:";
  cin>>b;

   fact=factorial(b);

   cout<<"\n Factorial of "<<b<<" is "<<fact;
 } /* end of main */

long int factorial(int x)

 {
    long int f=1;
    int i;

    for(i=x;i>=1;i--)
    f=f*i;

    return(f);

 }
```

Output

Enter number : 25
factorial of 25 is 2076180480

Problem 3.28

Write a program in C++ to input two numbers and operation symbol (+, –, * or /) and perform operation as per symbol.

Program

```
// 3.28 Operation as per symbol

# include<iostream.h>

void arithop(float a,float b,char op);

void main()

{
  float num1,num2;
  char opn;

  cin>>num1>>num2>>opn;

  cout<<"\n"<<num1<<opn<<num2<<"=";

  arithop(num1,num2,opn);

} /* end of main */

void arithop(float a,float b,char op)

{

  switch(op)

  {
    case'+':
         cout<<a+b;
         break;

    case'-':
         cout<<a-b;
         break;

    case'*':
         cout<<a*b;
         break;

    case'/':
         cout<<a/b;
         break;

    default :
```



```
         cout<<"invalid operation\n";
    }


  }
```

Output

34 565 +
34 + 565 = 599
67 89 ^
67 ^ 89 = invalid operation.

Problem **3.29**

Develop a function in C++ which adds 'N' natural numbers. Call the function in main function.

Program

```
// 3.29 Sum of N natural numbers

# include<iostream.h>


void main()

{

  int n;
  int sum(int n);

  cout<<"\n enter value for n:";
  cin>>n;

  cout<<"sum of first "<<n<<" natural nos ="<<sum(n);

}


int sum(int m)

{
  int i,result=0;

  for(i=1;i<=m;i++)
   result+=i;

   return(result);

}
```

Output

```
Enter value for n : 100
sum of first 100 natural nos = 5050
```

Problem 3.30

Write a program in C++ to find maximum of any three numbers using multiple return statements in a function definition.

Program

```
// 3.30  Using multiple return

# include<iostream.h>

void main()

{
 float maximum(float,float,float);
 float x,y,z,max;
 cout<<"enter three numbers\n";
 cin>>x>>y>>z;
 max=maximum(x,y,z);
 cout<<"maximum="<<max;

}
float maximum(float a,float b,float c)
{
  if(a>b)
   {
     if(a>c)
        return(a);
      else
        return(c);
    }
    else
       {
          if(b>c)
          return(b);
          else
          return(c);
  }
}
```

Output

```
Enter three numbers
4     5     6
maximum = 6
```


Problem 3.31

Write a program in C++ to find sum of the following series using a function declaration.

$$\text{sum} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \text{-----} \frac{(x^n)}{n!}$$

where x and n are entered from the keyword.

Program

```
// 3.31 Sum of the series
# include <iostream.h>
void main()
{
    long int fact(int);
    float power(float,int);
    float sum,temp,x,pow;
    int sign,i,n;
    long int factval;

    cout<<"Enter a value for n ?"<<endl;
    cin>>n;
    cout<<"Enter a value for x ?"<<endl;
    cin>>x;

    i=3;
    sum=x;
    sign=1;
    while(i<=n)
    {
        factval=fact(i);
        pow=power(x,i);
        sign=(-1)*sign;
        temp=sign*pow/factval;
        sum=sum+temp;
        i=i+2;
     }
     cout<<"Sum ="<<sum;
 }
long int fact(int max)
{
  long int value;
  value=1;
  for(int i=1;i<=max;++i)
    {
        value=value*i;
```

```
    }
    return(value);
}
float power(float x,int n)
{
   float value2;
   value2=1;
   for(int j=1;j<=n;++j)
      value2=value2*x;
      return(value2);
}
```

Output

Enter value for n ?
7
Enter a value for x ?
1
sum = 0.841468

Problem 3.32

Write a functions to compute area of circle and volume of sphere using formulas :

Area of circle = πr^2

Volume of sphere = πr^3.

Accept input radius and compute area of circle and volume of sphere.

Program

```
// 3.32  Area of circle and Volume of sphere

# include <iostream.h>
# include <math.h>

const double PI=3.14159;

double area(double radius);
double volume(double radius);

void main()
{
   double radius,a,v;

   cout<<"Enter Radius (in inches):";
   cin>>radius;

   a=area(radius);
   v=volume(radius);
```



```
   cout<<"\n Radius="<<radius<<" inches\n"
       <<"Area of circle="<<a<<" square inches\n"
       <<"Volume of sphere="<<v<<" cubic inches\n";
}
double area(double radius)
{
   return(PI*pow(radius,2));
}

double volume(double radius)
{
   return((4.0/3.0)*PI*pow(radius,3));
}
```

Output

Enter Radius (in inches) = 2
Radius = 2 inches
Area of circle = 12.56636 square inches
Volume of sphere = 33 .510 293 cubic inches

Recursion

In C++, it is possible for the functions to call themselves. A function is called recursive if a statement within the body of function calls the same function. Recursion is the process of defining something in terms of itself. In recursion a process of chaining occurs. Recursion sometimes is also called as circular definition.

In some cases, Recursion is an efficient programming technique. It reduces the size of code. A very simple example of recursion is recursive function factr ( ), which computes factorial of a number. The factorial of a number n is the product of all whole numbers between 1 and n. For example 4 factorial is $1 \times 2 \times 3 \times 4$ i.e. 24. The factr ( ) function is shown below.

```
int factr (int n)
 {
    int ans ;
    if (n = = 1) return (1) ;
    ans = factr (n - 1) * n ;
    return (ans) ;
 }
```

The statement ans = factr (n–1) * n is a recursive statement because it is defined in body of factr ( ) function itself.

The operation of recursive function is bit complex to understand.

When factr () is called with an argument of 1, the function returns 1, otherwise it returns the product of factr (n – 1) * n. To evaluate this expression factr is called with n–1. This continues to happen until n equal 1.

```
    }
  return(value);
}
float power(float x,int n)
{
  float value2;
  value2=1;
  for(int j=1;j<=n;++j)
    value2=value2*x;
    return(value2);
}
```

Output

Enter value for n ?
7
Enter a value for x ?
1
sum = 0.841468

Problem 3.32

Write a functions to compute area of circle and volume of sphere using formulas :

Area of circle = πr^2

Volume of sphere = πr^3.

Accept input radius and compute area of circle and volume of sphere.

Program

```
// 3.32  Area of circle and Volume of sphere

# include <iostream.h>
# include <math.h>

const double PI=3.14159;

double area(double radius);
double volume(double radius);

void main()
{
  double radius,a,v;

  cout<<"Enter Radius (in inches):";
  cin>>radius;

  a=area(radius);
  v=volume(radius);
```



```
  cout<<"\n Radius="<<radius<<" inches\n"
      <<"Area of circle="<<a<<" square inches\n"
      <<"Volume of sphere="<<v<<" cubic inches\n";
}
double area(double radius)
{
  return(PI*pow(radius,2));
}

double volume(double radius)
{
  return((4.0/3.0)*PI*pow(radius,3));
}
```

Output

Enter Radius (in inches) = 2
Radius = 2 inches
Area of circle = 12.56636 square inches
Volume of sphere = 33 .510 293 cubic inches

Recursion

In C++, it is possible for the functions to call themselves. A function is called recursive if a statement within the body of function calls the same function. Recursion is the process of defining something in terms of itself. In recursion a process of chaining occurs. Recursion sometimes is also called as circular definition.

In some cases, Recursion is an efficient programming technique. It reduces the size of code. A very simple example of recursion is recursive function factr ( ), which computes factorial of a number. The factorial of a number n is the product of all whole numbers between 1 and n. For example 4 factorial is $1 \times 2 \times 3 \times 4$ i.e. 24. The factr ( ) function is shown below.

```
int factr (int n)
{
    int ans ;
    if (n = = 1) return (1) ;
    ans = factr (n - 1) * n ;
    return (ans) ;
}
```

The statement ans = factr (n–1) * n is a recursive statement because it is defined in body of factr ( ) function itself.

The operation of recursive function is bit complex to understand.

When factr () is called with an argument of 1, the function returns 1, otherwise it returns the product of factr (n – 1) * n. To evaluate this expression factr is called with n–1. This continues to happen until n equal 1.

Suppose n = 3, the sequence of operations which are performed are shown below.

```
ans  = factr (2) * 3
     = factr (1) * 2 * 3
     = 1 * 2 * 3
     = 6
```

When a function calls itself, a new set of local variables and parameters are allocated storage on stack, and function code is executed from the top with these new variables.

As recursive call returns, the old local variables and parameters are removed from stack and execution resumes at the point of the function call inside the function.

Most of recursive codes do not significantly reduce code size. Also they are executed bit slower. The main advantage of recursion is that you can use them to create clear and simple version of several algorithms. Problems related to Artificial intelligence are easy to code with recursion.

When we use recursion, we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.

Problem 3.33

Write a program in C++ to generate Fibonacci sequence by recursion.

The Fibonacci sequence is
1 1 2 3 5 8 13 -----

Program

```
// 3.33  Fibonacci sequence by recursion

# include<iostream.h>

void main()

{
  int n;
  void fib(int n);

  cout<<"number of terms to be generated ?\n";
  cin>>n;
  cout<<"\n\nFibonacci sequence upto "<<n<<"
terms:\n\n";

  fib(n);

}

void fib(int n)

{
  static int f1=0,f2=1;
```



```
int temp;

if(n<2)
   {
     f1=0;
     f2=1;
     }

  else
  {
    fib(n-1);
    temp=f2;
    f2=f1+f2;
    f1=temp;
  }

   cout<<f1<<"  ";

    return;

 }
```

Output

Number of terms to be generated
fibonacci sequence upto 4 terms :
1 2 3 5

Problem 3.34

Write a program in C++ to find greatest common divisor (GCD) of two integers.

Program

```
// 3.34 Gcd of two numbers

# include<iostream.h>

void main()

{
    int a,b,gcd;
    int hcf(int a,int b);

    cout<<"\n type in any 2 numbers \n";
    cin>>a>>b;

    gcd=hcf(a,b);

    cout<<"\n GCD of "<<a<<" and "<<b<<" is :"<<gcd;
```

```
}

int hcf(int p,int q)

{
    int r,factor;

    r=p-(p/q*q);

    if(r==0)
      return(q);

      else hcf(q,r);

}
```

Output

Type in any 2 numbers

5 15

GCD of 5 and 15 is : 5

Function overloading

In real life, many times, we use the same name to perform similar actions. For example, whether we are determining absolute value of an integer or float or double, we call it as determination of absolute value of a number. As we are aware that if the type of the number changes, the actions to be performed gets differed, but overall general purpose of actions remains same.

Suppose we are determining minimum of numbers. You may write a set of actions which determines minimum of 2 numbers, another set of actions which determines minimum of 3 numbers and yet another set of actions which determines minimum of 4 numbers. The set of actions is different, but over all their general purpose is same.

Is it possible to use the same name for similar actions? Yes, C++ allows you to give two or more different function definitions the same function name, and it is called function overloading. This is the advantage with C++. In rest of the programming languages you will be needed to use seperate function name for seperate function definition. For example, we may be needed to use three seperate functions names for determination of absolute value of integer, long and double as shown below.

```
int abs (int i) ;
long labs (long l) ;
double dabs (double d) ;
```

All these functions will do the same thing. They have difference in their arguments. The first function abs ( ) have integer argument, the second function labs ( ) have long argument and third one dabs ( ) have double argument. It seems unnecessary to have different function names for same thing. C++ overcomes the situation by allowing the programmer to create three different functions with the same name. This is function overloading. The following program illustrates function overloading.


```
# include < iostream.h >
// abs is overloaded three ways
int abs (int i) ;
long labs (long l) ;
double dabs (double d) ;
void main ( )
  {
    cout << abs (-10) << "\n" ;
    cout << abs (-11.0) << "\n" ;
    cout << abs (-9L) << "\n" ;
  }
int abs (int i)
  {
    cout << "Using integer abs ( ) \n" ;
    return (i < 0 ? - i : i) ;
  }
long abs (long l)
  {
    cout << "Using long abs ( ) \n" ;
    return (l < 0 ? - l : l) ;
  }
double abs (double d)
  {
    cout << "Using double abs ( ) \n" ;
    return (d < 0 ? - d : d) ;
  }
```

The output of the program is :

```
Using integer abs ( )
   10
Using double abs ( )
   11
Using long abs ( )
   9
```

Here three functions are having same name. The compiler differentiates between these three on the type of the argument. i.e. the compiler calls the appropriate function based on the type of the argument. Observe that all three functions have different type of argument.

Overloading is great concept. It makes the program easier to read. It saves your energy to think for a new name every time. You can use the natural names for function definitions. This makes programming closer to real life.

In general, to over load a function, simply declare different versions of it. The compiler takes care of the rest. One important restriction in overloading is that the type and/or number of parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types.

Also you should note that you can use the same name to overload only related functions and not unrelated functions. For example you can use name sqr ( ) to create functions that returns square of an int and square root of a double. However, these two operations are different, and purpose of overloading is not served here. i.e. You should overload only similar operations.

Default Arguments

When we call a function, we need to pass values to the arguments of function. If they are not passed, there is possibility of getting assigned some garbage values. C++ allows us to define default values for the arguments when the values are not passed when the function call is made. The default value can be specified in manner similar to that of variable initialization.

For example, for the function funct ( ) we can declare a default value for argument d as shown below.

```
Void funct (double d = 0.0)
 {
    -----
    -----
 }
```

Now the function funct ( ) can be called in two ways.

```
funct (201.01) ;
funct ( ) ;
```

In first call we are passing an explicit value of 201.01 to the argument d. The default value will not be considered here.

In second call, we are not passing an explicit value to the argument d. Here the call gives d automatically the default value 0.0

The default arguments makes writing programs easier. A programmer may need to handle variety of situations. In most of the situations the argument may have a common value. You can specify a default value which is a common value. So programmer need not pass value explicitly to the argument every time. Only for incommon situations he will be needed to pass value to the argument explicitly. Suppose we are writing a function clrscr ( ) function which clears the screen. A very common video mode displays 25 lines of text. In such a case we can provide a default value of 25 to the argument as shown below.

```
void clrscr (int size = 25)
 {
    for (; size ; size --)
      cout << end l ;
 }
```

When we make a call to function clrscr ( ) as :

```
clrscr ( ) ;
```

It clears 25 lines on screen

The call

```
clrscr (10) ;
```

Will clear only 10 lines on screen. Here we are passing explicit value of 10 to argument size. Observe that when default value is appropriate we need not specify the value.

When there are more than one arguments, then only trailing arguments can have default values. That is we must add defaults from right to left. We can not provide a default value to the argument in the middle of the list.

For example,

```
float amount (float principal, int peroid, float rate = 0.15);
```

This declares on default value 0.15 to argument rate.

A function call

```
value = amount (5000, 7) ;
```

Passes an explicit value 5000 to principal, 7 to peroid and lets function use the default value 0.15 for rate.

The declaration

```
float amount (float principal, int peroid = 5, florat rate) ;
```

is invalid while the declaration

```
float amount (float principal, int peroid=5, float rate=0.15);
```

is a valid declaration.

Inline functions

The advantage of using functions is that they save memory space. As the calls to the function causes the same code to be executed, the function body need not be duplicated in memory.

When a function is called, certain amount of time is spent on passing values, passing control, returning values and returning control. To save this time, we need to instruct C++ compiler to put code in function body directly inside the program. So whenever function is called, the actual code from the function would be inserted, instead of a jump to the function. Such functions are inline functions.

An inline function is a function that is expanded in line when it is invoked, i.e. compiler replaces the function call with corrosponding function code. The inline function has the form :

```
inline function - header
{
        function body
}
```

For example,

```
inline double cube (double a)
 {
    return (a * a * a) ;
 }
```

This is a small function which returns cube value of a. We can use this function in main ( ) as,

```
main ( )
  {
    cout << cube (3.0) ;
  }
```

Whenever inline function is called, the code of function is inserted by compiler. The above program is equivalent to.

```
main ( )
  {
    cout << 3.0 * 3.0 * 3.0 ;
  }
```

The inline functions in C++ are important because they allow us to create very efficient code. But they are useful for small size code. Also we inline only those functions that will have significant impact on the performance of your program.

Observe that inline is not a command but a request to compiler. The compiler may ignore it. One has to see compiler's user manual for restrictions on inline. For example a compiler may not inline a recursive function.

Header files

Header files are files with .h extension. These files are available with compiler. These files consist of code for library functions. Whenever we use these functions in program, we need to include corrosponding header file.

For example, `# include <isotream.h>`

Some commonly used header files are listed below.

Sr.No.	Header File	What it does
1.	alloc.h	Declares memory management functions (allocation, deallocation etc.)
2.	assert .h	Defines the assert debugging macro.
3.	bios .h	Declares the various functions used in calling IBM PC ROM BIOS routines.
4.	complex .h	Declares complex math functions.
5.	conio .h	Declares various functions used in calling DOS console I/O routines.
6.	ctype.h	Contains information used by character classification and character conversion macros.
7.	dir. h	Contains structures, macros and functions for working with directories and path names.
8.	dos .h	Defines various constants and give declaration needed for DOS and 8086 specific calls.
9.	errno. h	Defines constant mnemonics for the error codes.
10.	float . h	Contains parameters for floating point routines.

Sr.No.	Header File	What it does
11.	graphics. h	Declares to prototypes for the graphics functions.
12.	io.h	Contains structures and declarations for low level input/out put routines.
13.	math.h	Declares prototypes for the math functions, defines the macro HUGE-VAL and declares the exception structure used by metherr.
14.	mem.h	Buffer manipulation functions
15.	search. h	Declare functions for searching and sorting.
16.	stdarg. h	Defines macros used for reading the argument list in functions declared to accept a variable number of arguments.
17.	stdef. h	Defines several common datatnpes and macros.
18.	stdio.h	Defines types and macros needed for standard I/O package Defines the standard I/O predefined streams stdin, stdout, stdprn, stderr. Declares stream level I/O routines.
19.	stdlib.h	Declares several commonly used routines- conversion routines, search/sort routines, etc.
20.	string.h	Declares several string and memory manipulation routines.
21.	sys\stat. h	Define symbolic constants used for opening and creating files.
22.	time. h	Defines a structure filled in by the time-conversion routines, and a type used by other time routines; also provide prototypes for these routines.
23.	iostream. h	Declares the basic C++ streams (I/O) routines.
24.	iomanip. h	Declares the C++ streams I/O manipulators and contains macros for creating parameterized manipulators.
25.	fstream. h	declares the C++ stream classes that support file input and output.
26.	generic. h	Contains macros for generic class declarations.
27.	bcd. h	Declares the C++ class bcd and the overloaded operators for bcd and bcd math functions.
28.	stdiostr. h	Declares the C++ stream classes for use with stdio FILE structures.

Standard library functions

Some of the standard C and C++ functions are given below. Whenever required, you can use a C- function in C++ code as well.

Function	Use	Name of Header File Containing Function
1. Data conversion function:		
atof	Convert string to float	stdlib.h
atoi	Convert string to int	stdlib.h
atol	Convert string to long	stdlib.h
ecvt	Converts double to string	stdlib.h
fcvt	Converts double to string	stdlib.h
gcvt	Converts double to string	stdlib.h
itoa	Converts int to string	stdlib.h
ltoa	Convert long to string	stdlib.h
strtod	Convert string to double	stdlib.h
strtoul	Convert string to unsigned long integer	stdlib.h
ultoa	Convert unsigned long to a	
2. Character classification functions		
isalnum	Tests for alphanumeric character	ctype.h
isalpha	Tests for alphabetic character	ctype.h
isdigit	Tests for decimal digit	ctype.h
islower	Tests for lowercase letters	ctype.h
isspace	Tests for white space character	ctype.h
isupper	Tests for upper case character	ctype.h
isxdigit	Tests for hexadecimal digit	ctype.h
tolower	Tests character and converts to lower case if upper case	ctype.h
toupper	Tests character and converts to uppercase if lower case.	ctype.h
3. String Manipulation Functions		
strcat	appends one string to another	string.h
strchar	Finds first occurrence of a given character in a string	string.h
strcmp	Compares two strings	string.h
strcmpi	Compare two strings without regards to case	string.h
strcpy	Copies one string to another	string.h
strdup	Dupricates a string	string.h
strlen	Finds length of a string	string.h
strlwr	Converts a string to lower case	string.h
strncat	Appends a portion of one string to another	string.h
strncmp	Compartes a portion of one string with a prtion of another string.	string.h
strncpy	Copies a given number of characters of one string to another	string.h
strrchr	Finds last occurance of a given chracter in a string	string.h

strrev	Reverses a string	string.h
strset	Set all characters in a string to a given character	string.h
strstr	Finds first occurance of a given string in another string	string.h
strupr	Converts a string to upper case	string.h
4. Searching and sorting functions		
bsearch	Performsbinary search	search.h
lfind	Performs linear search for given value	
qsort	Performs quick sort	
5 I / O functions		
close	Closes a file	io.h
fclose	Closes a file	stdio.h
feof	Detets end of file	stdio.h
fgetc	Reads a character from file	stdio.h
fgetchar	Reads a character from keyboard	stdio.h
fgets	Reads a string from file	stdio.h
fopen	opens a file	stdio.h
fprintf	writes formatted data to a file	stdio.h
fputc	writes a character to file	stdio.h
fputchar	writes a character to screen	stdio.h
fputs	writes a string to a file	stdio.h
fscanf	Reads formatted data from a file	stdio.h
fseek	Repositions file pointer to given location	stdio.h
fteel	Gets current file pointer position	stdio.h
get c	Reads a character from file	stdio.h
getch	Reads a character from keyboard	stdio.h
getche	Reads a character from key board and echoes it.	stdio.h
gets	Reads line from key board	stdio.h
open	opens a file	io.h
printf	writes formatted data to screen	stdio.h
putc	writes character to file	stdio.h
putch	writes character to s screen	stdio.h
puts	write a line to file	stdio.h
read	reads data from file	stdio.h
rewind	repositons file pointer to the begining of a file	stdio.h
scanf	reads fromatted data from keyboard	stdio.h
sprintf	writes form atted output to a string	stdio.h
tell	gets current file pointer position	stdio.h
write	writes data to a file	String.h
strncpy	Copies a given number of characters	stdio.h
write	writes data to a file	

6. File handling functions		
remove	Delete files	stdio.h
rename	Renames files	stdio.h
unlink	Delete files	stdio.h
7. Directory control functions		
ondir	changes curoent working directory	dir.h
getcwd	Gets current working directory	dir.h
fnsplit	Splits a full path name in to its components	dir.h
findfirst	Searches a disk directory	dir.h
findnext	Continues find first search	dir.h
mkdir	Makes a new directory	dir.h
rmdir	Removes a directory	dir.h
8. Buffer manipulation functions		
memchr	Return a pointer to the first occurance, within a specified number of characters, of a given character in buffer	mem.h
memcmp	Compares a specified number of charaters from two buffers.	mem.h
memcpy	copies a specified number of characters from one buffer to onother.	mem.h
memicmp	Compares a specified number of characters from two buffers without regard to cse of the letters.	mem.h
memmove	Copies a specified number of characters from one buffer to another	mem.h
memset	Uses a given character to initialize a specified number of bytes in the fuffer	mem.h
9. Disk I / O Functions		
absread	Reads alsolute disk sectors	dos.h
abswrite	Writes absolute disk sectors	dos.h
bopdfodk	Bios disk services	dos.h
getdisk	gets current drive number	dos.h
setdisk	Sets current disk drive	dos.h
10. Memory Allocation Functions		
calloc	Allocates a block of memory	alloc.h
farmalloc	Allocates memory from for heap	alloc.h
farfree	frees a block from for heap	alloc.h
free	frees a block allocated with malloc	alloc.h
malloc	Allocates a block of memory	alloc.h
realloc	Reallocates a block of memory	alloc.h

11. Process Control Functions		
abort	Aborts a Process	process.h
atexit	Executes function at program termination	process.h
execl	Executes child process with argument list	process.h
exit	Terminates the process	process.h
spawnl	Executes child process with argment list	process.h
spawnep	Executes child processing PATH variable & argument list.	process.h
system	Executes an MS Dos commond	process.h
12. Graphics functions		
arc	Draws an arc	graphics.h
ellipse	Draws an ellipse	graphics.h
floodfill	Fills an area of the screen with current color	graphics.h
getimage	stores a screen imaqe in memory	graphics.h
getlinestyle	oblains the current line style	graphics.h
getpixel	obtains the pixels value	graphics.h
lineto	Draws a line from the currant graphic output position to the specified point	graphics.h
moveto	Moves the current graphic out put position to a specified point	graphics.h
pieslice	Draws a pie-slice- shaped figure	graphics.h
putimage	Retrieves an image from memory and displays it	graphics.h
rectangle	Draws a rectangle	graphics.h
setcolor	sets the current color	graphics.h
setlinestyle	sets current line style	graphics.h
putpixel	sets a pixel's value	graphics.h
setviewport	Limits graphic output and positions the logical origin within the limited area	graphics.h
13. Time related functions		
clock	Returns the elapsed cpu time for a process	time.h
dirfftime	Computes the difference between two times	time.h
ftime	Gets current system time as structure	time.h
strdate	Returns the current sytem date as a string	time.h
strtime	Returns the current system time as a string	time.h
time	Gets current system time as long intger	time.h
setdate	Sets DOS date	time.h
getdate	Gets system date	time.h
14. Miscellaneous Functions		
delay	suspends execution for an interval (milliseconds)	dos.h
random	Generate random numbers	stdlib.h
randomize	Initializes random number generation with a random value based on time	stdilb.h
sound	Turns PC speaker on at a specified freguency	dos.h
nosound	Turns PC speaker off	dos.h

15. Dos interface functions

FP_ OFF	Returns offset portion of a far pointer	dos.h
FP_SEG	Returns segment portion of a far pointer	dos.h
getvect	Gets current value of specified interrupt vector	dos.h
keep	Installs terminate and stay resident (TSR) programs	dos.h
int86	Invokes MS DOS interrupts	dos.h
int86x	Invokes MS DOS interrupts with segment register values	dos.h
intdosx	Invokes MS-DOs service with segment register values	dos.h
MK_FP	Makes a far pointer	dos.h
segread	Returns current values of segment registers.	dos.h
stetvect	Sets current value of specified interrupt veetor	dos.h

Programming Examples

Problems 3.35

Write a program in C++ which determines volume of a cube, cylinder and rectangular box using following formulas.

Volume of cube = s^3.
volume of cylinder = $\Pi r^2 h$.
volume of rectangular box = lbh_1.
Assume s = 10, Π = 3.14519, r = 2.5 h = 8, l = 100, b = 75 and h_1 = 15
Use a overloaded function volume ( ).

Program

```
// 3.35  Function Overloading

# include <iostream.h>

int volume(int);
double volume(double,int);
long volume(long,int,int);

void main()
{
   cout<<volume(10)<<"\n";
   cout<<volume(2.5,8)<<"\n";
   cout<<volume(1001,75,15);
}

int volume(int s)
{
```



```
   return(s*s*s);
}

double volume(double r,int h)
{
   return(3.14519*r*r*h);
}

long volume(long l,int b,int h)
{
   return(l*b*h);
}
```

Output

```
1000
157.2595
112500
```

QUESTIONS - 3.3

1. What are functions ?
2. What are types of functions ?
3. What are advantages of functions ?
4. What is function prototyping ? What are its advantages ?
5. Describe the general form of function definition.
6. Describe call be value.
7. What is purpose of return statement ? What are different forms of return statement ?
8. What types of values a function can return ?
9. What is scope ?
10. Differentiate between :-
    i) Local and Global variables.
    ii) Actual and formal variables.
    iii) Global and external variables.
11. What are static variable ? Explain its values.
12. What are advantages of register variables ?
13. What is recursion ?
14. What are advantages and limitations of recursion ?
15. What is overloading ?
16. What is function overloading ? What are its advantages ?
17. How to define default values for the arguments of functions ?

18. In what situations default arguments are useful ?
19. What are inline functions ? What are advantages of inline functions ?
20. What are header files ? Describe purpose of header files in C ++
21. Write a program in C++ that will transform cartesion coordinates into spherical coordinates using the formula.

$$r = \sqrt{x^2+y^2+z^2}$$

$$\theta = \tan^{-1}\left(\sqrt{x^2+y^2}/z\right)$$

and $\theta = \tan^{-1}(y/x)$

22. Write a function in C++ to find the odd and even numbers from a given set of numbers.
23. Write a function in C++ to find the sum of the following series.
    i) sum = 1 – 3 + 5 – 7 ----- n
    ii) sum = 1 – (1/2 !) + (1/3 !) + ... (1/n !)
23. Write a program in C++ to print table of values for the functions.

$$\sin h(x) = \frac{e^x - e^x}{2}$$

and $\cos h(x) = \frac{e^x + e^{-x}}{2}$

for x = –1 to 1 in increments of 0.1.

25. Two positive integers are said to be "buddies" if each one is equal to the sum of the divisors of the other and the divisors include 1 but exclude the number itself.
    For example. the numbers 220 and 284 are buddies since :
    divisors of 220 = 1 + 2 + 4 + 5 + 10 + 20 + 22 + 44 + 45 + 110 = 284
    divisors of 284 = 1 + 2 + 4 + 71 + 142 = 220
    Write a program to find all the pairs of buddies in a given range of numbers.

Pointers

Pointer is important feature of C++ language. In C++ it is frequently used. It is a powerful and handy tool once it is mastered. Using of pointer provides certain advantages.

1. Pointer allows to pass variables, arrays, functions, strings and structures as function arguments.
2. A pointer allows to return structured variables from functions.
3. It supports dynamic allocation and deallocation of memory segments.
4. With the help of pointers, variables can be swapped without physically moving them.
5. It allows to establish a link between data elements or objects.

Pointers are one of the strongest but also one of the dangerous features in C++. It may cause your system to crash if they are not utilized properly. If they are used incorrectly, it causes bugs that are difficult to find.

What is Pointer ?

As we know computers uses memory for storing instructions and the values of variables. The computer memory is a sequential collection of storage cells. Each cell has a number called address of the cell.

Whenever we declare a variable, then it gets associated with certain location where the value of the variable is stored.

Consider the declaration

```
int i = 30 ;
```

This statements tells C++ compiler to :

i) Reserve space in memory to hold integer value.

ii) Associate the name of the variable i with this location.

iii) Store the value 30 at this location.

Figure 5.1 show the variable declaration.

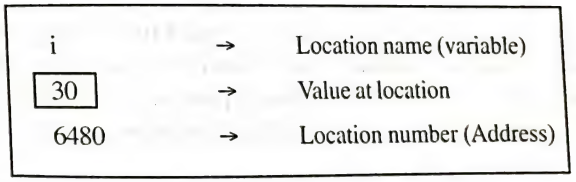


Fig. 5.1 : Variable declaration.

We observe that the computer has selected a memory location 6480 as a place to store value 30. The location is changing, as some other time computer may choose another location to store value. Here the address of i in memory is 6480.

The value 30 can be accessed by either name i or the address 6480. This memory address can be assigned to some variable. Such variable which holds memory address are called pointers.

A pointer is a variable that holds a memory address. This address is the location of another object (typically variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second. Figure 5.2 illustrates this.

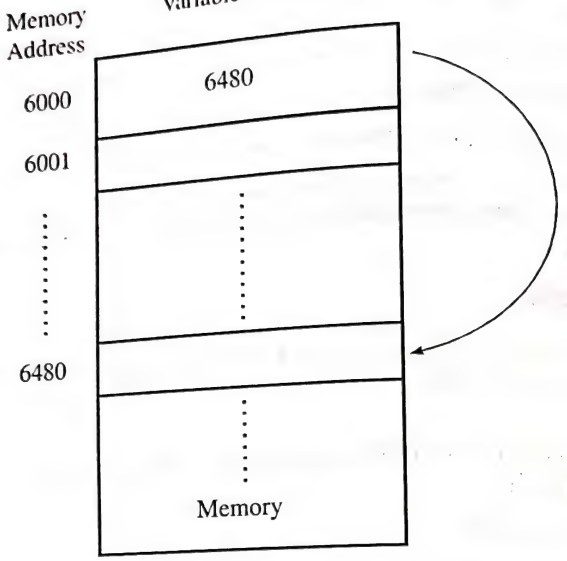


Figure 5.2 : Pointer as a variable

Observe that pointer is a variable whose value is also stored in memory in another location.

Declaring a pointer variable

In C++, every variable must be declared for its type before use. Similarly the pointer variable must be declared before we use them. The declaration of pointer variable has the form :

Data type * pointer _ name ;

Here data type is base type of the pointer and may be any valid type. The pointer_name is name of the pointer variable. The asterisk * tells that pointer_name is a pointer variable. The data type of pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point to anywhere in memory. However, all operations to pointer are relative to its base type.

For example, `int *p;`

This statement declares that p is a pointer variable that points to an integer data type. Observe that the type int refers to the data type of variable being pointed to by p and not the value of the pointer.

The Pointer operators

There are two special pointer operators : *and &. The & is unary operator that returns the memory address of its operand.

For example,

```
m = &count;
```

assigns the memory address of variable count to m. This address is the location address of variable. The operator & can be remembered as "the address of". Therefore the above statement means "m receives the address of count".

If the variable count uses the address say, 6500 to store its value, say 10, then after above assignment, m will have a value 6500.

The second pointer operator is *. This is also a unary operator called as indirection operator. It returns the value located at the address that follows. For example, if m contains the memory address of the variable count,

```
q = *m;
```

places the value of count into q. Thus q will have a value 10 because 10 is stored at location 6500, which is memory address that was stored in m. The operator * can be remembered as "value at address". The above statement means, "q receives the value at address m".

Both & and * have higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

```
m = &count;
q = *m;
```

are equivalent to

```
q = *& count ;
```

The following program demonstrates the above.

```
void main ( )
{
   int count = 10 ;
   int *m ;
   m = & count ;
   cout << " \n Address of count = " << & count ;
   cout << " \n Address of count = " << m ;
   cout << " \n Address of m = " << & m ;
   cout << " \n Value of m = " << m ;
   cout << " \n value of count = " << count ;
   cout << " \n value of count = " << *& count ;
   cout << " \n value of count = " << *m ;
}
```

The output of above program is :

```
Address of count = ox8f9dffff4
Address of count = ox8f9dffff4
Address of m = ox8f9dffff2
value of m= ox8f9dffff4
value of count = 10
value of count = 10
value of count = 10
```

We can change the value of variable indirectly using a pointer and the indirection operator.

```
*m= 25;
```

will change value of count to 25.

Pointer Expressions

Pointer expressions are expression involving pointers. They follows same rules as other expressions. Let us discuss some aspects of pointer expressions.

(i) Pointer Assignment

We can use a pointer variable on right hand side of assignment statement to assign its value to another variable.

If p1 and p2 are pointer variables then we may write.

```
p2=   12;
```

Here both p1 and p2 will point to same variable.

(ii) Pointer Arithmetic

There are only two operations that you may use on pointers : addition and subtraction. Suppose p1 is an integer pointer with a current value 6000.

The expression,

```
p1 = p1 + p2;
```

makes p1 point to the twelfth element of p1's type beyond the one it currently points to.

If integers are 2 byte long, the news value of p1 will be 6024 and not 6012.

The expression,

```
p1 + + ;
```

will increment p1. The new value of p1 will be 6002 and not 6001.

The expression,

```
p1 - - ;
```

will decrement p1. The new value of p1 will be 5998.

Each time the pointer is incremented, it points to the memory location of the next element of its basetype. Each time it is decremented, it points to the location of the previous element. In case of character pointers it will cause increment and decrement by 1 only.

If p1 and p2 are pointers, then following statements

```
y =   *p1 * *p2 ;
sum =   sum + *p1 ;
*p2 =   *p2 + 10 ;
```

are all valid statements.

When we use / then between / and * we need to put one blank space otherwise it is treated as beginning of comment line.

```
z =   5 - *p2 / *p1 ;
```

You may subtract one pointer from another in order to find the number of objects of their base type that seperates the two. All other arithmetic operations are not allowed. You can not multiply or divide pointers, you may not add two pointers, you may not apply bitwise operators to them. The following statements are invalid statements,

```
p1 / p2 ;
p1 * p2 ;
p1 / 3 ;
p1 + p2 ;
```

All of these statements are not allowed.

Pointer Comparisions

You can compare two pointers in a relational expression If p1 and p2 are two pointers, then we can write

```
if (p1 > p2)
    cout << "p2 points to lower memory than p1" ;
```

Generally pointer comparisons are used when two or more pointers points to a common object, such as an array.

The following pointer comparisions can also be used.

```
p1==   p2
p1!=   p2
p1 <   p2
```

Pointers and Functions

Pointers can be used in function declaration. The complex functions can be easily represented and accessed by using pointers. When pointer variable is used as formal argument, then calling a function is referred as **call by reference**.

In call by reference, when a function is called by a program, the address of actual arguments are copied on to the formal arguments. That is, formal and actual arguments are referencing to same memory location. Therefore change in value of formal argument affects the value of actual argument. The call by reference is used when function produces more than one value and provides these values to the caller.

For example,

```
void main ( )
{
     void funct1 (int * x, int * y) ;
     -------------
     funct1 (& x, & y) ; // call by Reference
     -------------
}
void funct1 (int * x, int * y) ;
{
     -------------
}
```

We are calling the function in statement

```
funct1 (& x, & y) ;
```

and we are passing addresses of x and y to the formal pointer variables a and b. Therefore x and a refers to same memory location. Similarly y and b refers to same memory location. Any change in value of a or b will ultimately changes the value of x and y.

A function swap ( ) which exchanges the value of two integer variables using pointers as function arguments is shown below.

```
void swap (int * x, int *y)
{
  int temp ;
  temp = * x ;
  * x  = * y ;
  * y = temp ;
}
```

swap ( ) is called by passing the addresses of two variables whose values are to be interchanged.

```
swap (& i , & j) ;
```

passes address of i and j to pointer variable x and y.

swap ( ) function interchanges values of i and j.

Function Pointer

Function is not variable, it still has physical location in memory that can be assigned to a pointer. This address is entry point of the function and it is the address used when the function is called. Once the pointer points to the function. The function can be called through that pointer. Function pointer also allows functions to be passed as arguments to other functions.

The address of function can be obtained by using the function name without any parenthesis or arguments. This is similar to that we obtain arrays address by using only array name without indexes. For example when we write

```
p = strcmp ;
```

it returns address of function strcmp ( ) and assign it to pointer variable p.

The function pointer p can be declared as :

```
int (* p ) (const char * , const char *) ;
```

The return type of function pointer is int and it has two arguments declared as character pointers.

The following program uses check ( ) function which uses a function pointer.

```
void check (char * a, char * b, int (* cmp) (const char * ,
const char *)) ;
   void main ( )
   {
     char s1 [80] , s2 [80] ;
     int * p (const char *, const char *) ;
     p = strcmp ;
     gets (s1) ;
     gets (s2) ;
     check (s1, s2, p)
   }
     void check (char * a, char * b,
                 int (* cmp) (const char *, const char *))
     {
       cout << "Testing for equality \n" ;
       if (! (* cmp) (a, b))
                 cout << "Equal" ;
            else cout << "Not equal" ;
     }
```

Inside the check ( ) function, the expression

```
(* cmp) (a , b)
```

calls strcmp ( ), which is pointed to by cmp, with the arguments a and b.

References

C++ has a feature which is related to pointer is reference. A reference is an implicit pointer. A reference can be used three ways :

(i) as a function parameter.

(ii) as a function return value.

(iii) as a stand alone reference.

We will discuss each of these.

Reference as a Function Parameter

In C++ we can create a function that automatically uses call by reference parameter passing. This is done by using references.

In C++, there are two ways to achieve call by reference. First, you can explicitly pass a pointer to the argument. This is discussed in last section, and another is to use reference parameter. For most of the circumstance use of reference is a better technique. Let us see how to use this technique.

To create a reference parameter, precede the parameter's name with an &. For example, declaration of minus ( ) function with reference parameter is :

```
void minus (int & i) ;
```

When minus ( ) is called with another argument, i becomes another name for that argument. Any operations applied to i will affect the calling argument. i.e. i is an implicit pointer that automatically refers to the argument used in the call to minus ( ).

Once i is used like a reference, we need not to use * operator. Whenever we use i, it is implicit a reference to the argument and any changes made to i will affect argument.

Following program segment shows use of reference.

```
void minus (int & i) ; // i is a reference
void main ( )
 {
    int x ;
    x = 10 ;
    cout << x << " negated is " ,
    minus (x) ;    // No need to use &
    cout << x << "\n" ;
 }
    void minus  (int & i)
      {
        i = -i ;  // no need of *
      }
```


Observe that when we use reference parameter, it automatically refers to the argument used to call the function. Therefore the statement

```
i  = -i ;
```

operates on x. There is no need to use & operator to an argument. Inside the function we need not apply * operator.

In the last section while discussing call by reference we have written function swap ( ). The same function using reference parameter can be rewritten as follows :

```
void swap (int & i , int & i ) ;
void main ( )
 {
    int a, b ;
    a = 1 ; b = 2 ;
    cout << a << b ;
    swap (a, b) ;
    cout << a << b ;
 }
void swap (int & i, int & j)
    {
      int = t ;
      t = i ;
      i = j ;
      j  = t ;
    }
```

Observe that use of reference parameter increases readability of code and makes writing easier.

Returning References

A function may return a reference. This allows a function to be used on left side of an assignment statement. A function returning a reference can be declared as.

```
char & replace (int i)  ;
```

This function return a reference which points to char value.

The following program illustrates return of reference value.

```
char & replace (int i) ;
  char s [80] = "Hello Rahul" ;
    void main ( )
    {
          replace (5) = 'x' ;
          cout << s ;
    }
  char & replace (int i)
```

```
{
        return s [i] ;
}
```

Here replace (5) returns a reference to 5th element of string which is a blank space. The blank space is assigned to x.

The output of above program is, HelloxRahul

Independence References

We can declare a reference that is simply variable. A reference variable provides an alternative name (alias) for a previously defined variable. A reference variable can be created as :

```
data_type & reference_name = variable_name
```

For example,

```
float total = 105.30 ;
float & sum = total ;
```

Here total and sum refers to the same data object in memory. If we change value of total by

```
total = total + 10 ;
```

it will cause change in both the total and sum to 115.30.

Similarly sum = 0 ; will change value of both variables to zero.

A reference variable must be initialised at the time of declaration.

A reference variable just creates another name. It has a limited use in programs.

Memory Management Operators

C++ defines two unary operators new and delete, to perform task of allowcating memory dynamically at run time and freeing memory.

The new operator can be used to create objects of any type. It has the form :

```
pointer - variable = new data - type ;
```

The new operator produces a new nameless variable and returns pointer that points this variable. The new operator allocates sufficient memory to hold a data object of type data - type and returns the address of the object. The pointer - variable holds the address of memory space allocated.

For example,

```
p = new int ;
```

Here p is pointer of type integer. Here p is assigned an address of memory that is large enough to hold an integer.

When we write

```
* p = 100 ;
```

then a value of 100 is assigned to the memory. We can also initialize the memory using new operator. The general form of memory initialization is shown below :


```
pointer - variable = new data - type (value) ;
```

Here value specifies the initial value.

For example, we can write

```
p = new int (100) ;
```

which initializes memory to 100.

Here type of value must be compitable with the type of data for which memory is being allocated.

When the data object is not needed, it is destroyed to release the memory space for reuse. This is done by using operator delete.

The general form of delete is

```
delete pointer = variables ;
```

For example,

```
delete p ;
```

frees the memory.

The delete operator must be used only with a valid pointer previously allocated by using new.

Programming Examples

Problem 3.36

Write a program in C++ to demonstrate how a pointer to a function is declared to perform simple arithmetic operations such as addition, subtraction, multiplication and division of two numbers.

Program

```
//  3.36  Pointers to Functions

# include <iostream.h>

void main()
{
    float add(float,float); // Function declaration
    float sub(float,float);
    float mul(float,float);
    float div(float,float);
   float (*ptradd)(float,float); // Pointer to function declaration
    float  (*ptrsub)(float,float);
    float  (*ptrmul)(float,float);
    float  (*ptrdiv)(float,float);
    void menu(void);

    float a,b,value;
    char ch;

    ptradd=&add;
    ptrsub=&sub;
```

```
ptrmul=&mul;
ptrdiv=&div;

cout<<"Demonstration of pointer to functions\n";
cout<<"enter any two numbers \n";
cin>>a>>b;
menu();
while((ch=cin.get())!='q')
 {
   switch(ch)
{
  case 'a':
         value=(*ptradd)(a,b);
         cout<<"Addition of two numbers=";
         cout<<value;
         cout<<endl;
         break;
  case 's':
          value=(*ptrsub)(a,b);
          cout<<"Subtraction of two numbers=";
          cout<<value;
          cout<<endl;
          break;
  case 'm':
          value=(*ptrmul)(a,b);
          cout<<"Multiplication of two numbers=";
          cout<<value;
          cout<<endl;
          break;
  case 'd':
          value=(*ptrdiv)(a,b);
          cout<<"Divison of two numbers=";
          cout<<value;
          cout<<endl;
          break;
 }
 }
}

void menu()
{
  cout<<"a->addition \n "
      <<"s->subtraction \n"
      <<"m->multiplication \n"
      <<"d->divison \n"
      <<"q->quit \n"
```



```
      <<"option, please?\n";
}
float add(float x,float y)
{
  return(x+y);
}
float sub(float x,float y)
{
  return(x-y);
}
float mul(float x,float y)
{
  return(x*y);
}
float div(float x,float y)
{
  return(x/y);
}
```

Output

```
Demonstration of pointers to function enter any two numbers
10      20
a ->            addition
s ->            subtraction
m ->            multiplication
d ->            division
q ->            quit
option , please ?
d
Division of two numbers = 0.5
a
Addition of two numbers = 30
q
```

QUESTIONS - 3.4

1. What is pointer ?
2. What are advantages of using pointers ?
3. Give syntax for declaration of pointers.
4. What are pointer operators ?
5. What is pointer expression ?
6. What arithmetic operations can be preformed on pointers ?
7. Which relational operations can be performed on pointers ?
8. What is call by reference ?
9. What is function pointer ? What are its advantages ?
10. What are references ?
11. Describe use of reference as a function parameter.
12. Describe syntax of function returning reference.
13. What is reference variable ?
14. What are dynamic allocation operators in C++ ?
15. How memory is managed in dynamic allocation ?
16. Give syntax for defining pointer types.
17. Write a program in C++ to read a set of characters using a pointer and to print it in the reverse order.
18. Write a program in C++ to find number of vowels in each words of a given text using a pointer.
19. Differentiate between - call by value and call by reference.

Visual Basic

- Introduction to Visual Basic
- Visual Basic Environment
  - Menu bar
  - Tool bar
  - Tool box
  - Properties settings
  - Form layout
- Visual Basic Programming
  - Variables
  - Constants
  - Defining variables
  - Arrays
  - Relational operators
  - Control flow statements
  - Loop statements
  - Nesting of loops
  - Use of built in functions
  - Event driven programming
  - Simple VB project
  - simple calculator